

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

Server applications commonly organize a collection of concurrent threads, each of which executes the code necessary to process a request. This software architecture, which causes frequent context switches between unrelated pieces of code, decreases instruction and data locality and consequently reduces 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 computation throughput.

This paper addresses this problem through *cohort scheduling*, a new policy that increases data locality by batching the execution of similar operations arising from server requests. Effective implementation of this policy requires programming abstraction, *staged computation* which replaces threads. The *Staged Server* library provides an efficient implementation of cohort scheduling on a shared multiprocessor. Measurements of server applications written with this library show that cohort scheduling can improve server throughput by as much as 80% by reducing the processor cycle per instruction by 30% and cache misses by 50%.

1 Introduction

Server applications program to manage access to shared resources such as databases, mail stores, file systems, and web servers. Each receives a stream of requests, processes each, and produces a stream of results. Good server performance is important to minimize latency to access the resource and to constrain the server's ability to handle multiple requests. Commercial servers such as databases, have been the focus of considerable research to provide higher performance through hardware algorithms, and parallelism, as well as considerable development to improve the 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 are creating a growing performance bottleneck in many systems because of

slow loading of memory, which costs hundreds of cycles during which the processor is idle. Higher performance processors attempt to alleviate this performance mismatch through numerous mechanisms such as caches, TLBs, and branch predictors [27]. These mechanisms exploit well-known program property—spatial and temporal reuse of data—by keeping data that is likely to be used quickly in the processor's cache.

Server software often exhibits program locality and consequently achieves poor performance than other software. For example, many studies have found that commercial database systems running on-line transaction processing (OLTP) benchmarks can achieve rates of cache misses and instruction stalls which reduce processor performance to as low as 10% of its peak potential [20]. Part of this problem may be attributable to database systems' code size [28] but their execution models are also responsible.

These systems are structured to process threads in a strict order before invoking blocking operations and relinquishing control to processors executing in a different order. Non-looping code segments that exhibit little locality. For example, Barroso et al. compared TPC-B, OLTP benchmarks whose threads execute an average of 5 K instructions before blocking, gain TPC-D, compute-intensive decision support system (DSS) benchmarks whose threads execute an average of 1.7 M instructions before blocking [9]. On Alpha Server 4100, TPC-B achieves a hit rate of 8.9%, a miss rate of 7%, and an overall performance of 0.7 cycles per instruction (CPI) by contrast, TPC-D achieves a hit rate of 2% and a miss rate of 32% on the 62.

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

securitively execute a cohort of similar computation requests that arise on a server. Computations in a cohort are similar because they have the same processing requirements and data, and are executed consecutively. This improves program locality and increases hardware performance. *Staged computation* is a programming model that provides a programming abstraction by which a programmer can identify and group related computations and make explicit the dependencies that constrain scheduling. Staged computation moreover, has the additional benefits of reducing concurrency overhead and the cost of expensive error-prone synchronization.

We implement this scheduling policy in a programming model as a library (*StagedServer*). In our experiments, one with an I/O-intensive server and another with a compute-bound server, *StagedServer* performed significantly better than threaded versions. *StagedServer* lowered response time by a factor of 2.0, reduced CPU utilization by 30% and reduced cache misses by more than 50%.

The paper is organized as follows. Section 2 introduces cohort scheduling and explains its main advantages for program performance. Section 3 describes staged computation. Section 4 briefly describes *StagedServer*. Section 5 contains performance measurements. Section 6 discusses related work.

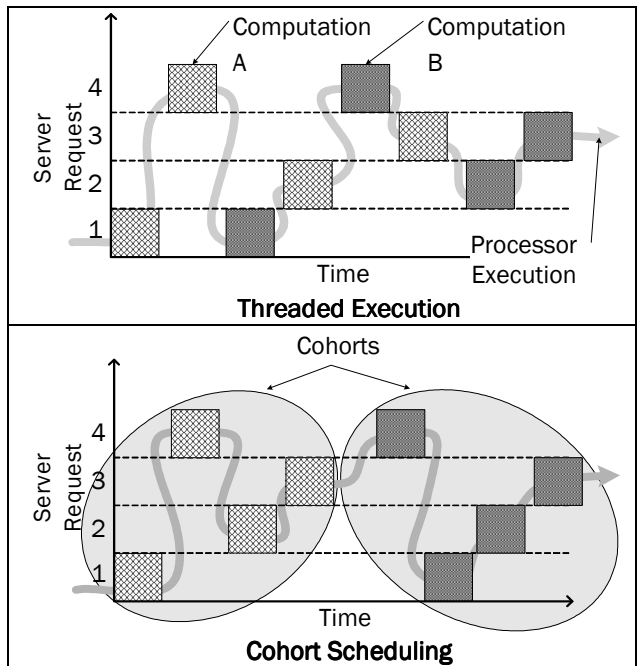


Figure 1. Cohort scheduling in operation. Shaded boxes indicate different computations performed while processing requests on a server. Cohort scheduling reorders the computations, so that similar ones execute consecutively on a processor, which increases program locality and processor performance.

2 Cohort Scheduling

Cohort scheduling is a technique for organizing the computation on a server application to improve program locality. The insight is that distinct requests on a server execute similar computations. Servers defer processing requests until a cohort of computations arrives at the processor. The cohort is then executed consecutively (Figure 1).

This scheduling policy increases opportunities for code reuse by reducing interleaving of unrelated computations that cause cache conflicts and evict each other. The approach is similar to loop tiling [19] which structures a matrix computation into submatrix computations that repeat references to the same submatrix. Cohort scheduling, however, dynamically reorganizes a set of computations into streams of similar computations of different items executed consecutively. The technique applies to uniprocessors and multiprocessors, so it depends on program locality to achieve good performance.

Figure 2 illustrates the results of an experiment that demonstrates the benefits of cohort scheduling on a multiprocessor. The speedup of executing different sized cohorts of synchronous random block I/O on a four-processor system whose cache and branch buffers had been flushed. As the cohort size increased, the speedup of each processor decreased rapidly. All processors consumed 109,000 cycles in the average cost dropped to 8% for cohorts of 8 and 82% for cohorts of 16. A direct measure of locality, cache misses, also improved dramatically. With cohorts of 16, cache misses per instruction dropped 77% from their initial value. The speedup declined 4% with cohorts of 16. These improvements required no changes to the operating system, only reordering operations in applications. Further improvements require reductions in self-conflict misses through better system calls, the amortizing roughly 500 cache misses.

2.1 Assembling Cohorts

Cohort scheduling is not a repairable technique for computation, but many benefits can be obtained if a programmer can explicitly form cohorts. For example, a program can transparently integrate cohort scheduling with threads. The basic idea is simple. A modified thread scheduler identifies and groups threads with identical program points (in PC space). Thread scheduling starts the program point likely to execute similar operations, with the behavior eventually diverging. The scheduler then schedules threads with identical PC before returning to the cohort.

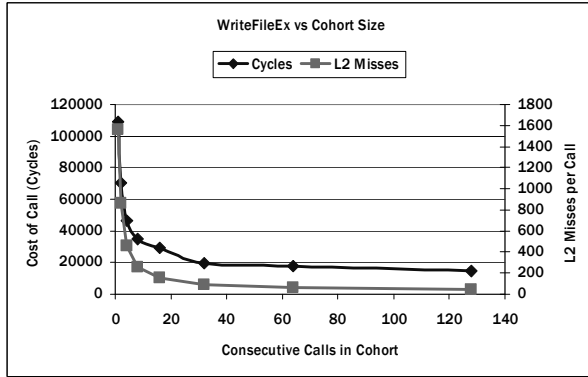


Figure 2. Performance of cohorts of WriteFileEx system calls in Window 2000 Advanced Server (Dell Precision 610 with an Intel Pentium III processor). The chart reports the cost per call—in processor cycles and L2 cache misses—of an asynchronous write to a random 4K block in a file.

It is believed that this scheme could sometimes improve performance and requires only minor changes to scheduler and changes to applications. However, it has a shortcoming in particular, on PC value are a coarse and indirect indicator of program behavior. Only threads with identical PC can be in a cohort, which misses many pieces of code with similar behavior. For example, several routines that access data structure might be in one cohort. Simple extension of this scheme such as the instance between PC as a measure of similarity has little to connect logical behavior and the perturbation of compilation and scheduling. Another disadvantage is that cohorts start after locking system calls rather than application-appropriate points in particular, compilation intensive application programs that are synchronous. It cannot be subject to lock.

To correct these shortcomings and properly assemble cohort, programmers must delineate computations and identify those that belong to a cohort. Staged computation provides programming abstraction that neatly captures both dimensions of cohorts.

3 Staged Computation

Staged computation is a programming abstraction intended to replace threads in concurrent or parallel programs. Stages offer compelling performance and correctness advantages and are particularly amenable to cohort scheduling. In this model, programs are 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, its invocation, execution, and reply are decoupled. Moreover, stages have scheduling autonomy which enables them to control their own concurrency with other operations to execute.

Stage is conceptually similar to a subject-based language, the tenth program structuring abstraction providing local and parallel operation. Stage however differs from objects in several respects. First, operation in stage is invoked asynchronously, that is, it does not wait for computation to complete, but instead continues and rendezvous later if necessary to retrieve a result. Second, stages autonomously control the execution of their operations. This autonomy extends to deciding when to execute the computation associated with invoked operation. Finally, stages are control abstraction used to organize and process work while objects are presentation oriented by their ties to function threads and 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 state and staff cohort scheduling only requires a simple scheduler that accumulates pending operations for a cohort.

Stages provide additional programming advantages as well. Because they control their own internal concurrency, they promote programming styles that reduce the cost of expensive error-prone explicit synchronization. Stages moreover provide a means for specifying and verifying properties of synchronous programs. This section briefly describes staged programming mode. Section 4 elaborates implementation in C++ library.

3.1 Stage Design

Programmers group operations in a stage for a variety of reasons. First, it is a means to regulate access to program state (e.g., static data) by wrapping abstract data type operations in a way that is obvious to a cohort, they typically have considerable instruction and data locality. Moreover, programmers control concurrency and reduce or eliminate synchronization in their programs (Section 4).

The second class of groupings is related to cooperation. Well-rounded and complete programming abstraction of this class are those compelling that first, but logically related operations frequently share code and data, collecting them in a stage identifies operations that would benefit from cohort scheduling.

This class encapsulate program control logic in the form of finite-state automata. As discussed below, stage's asynchronous operations are easily implemented as reactive transitions in an event-driven state machine.

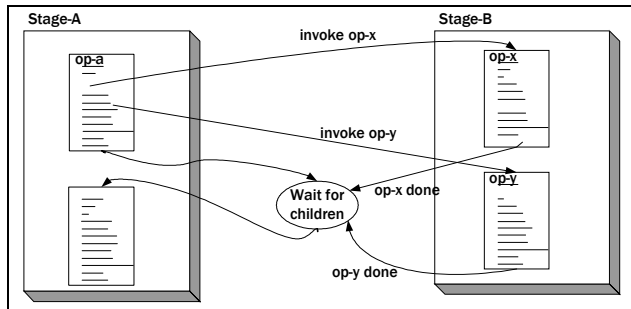


Figure 3. Example of stages and operations. Stage-A runs *op-a*, which invokes two operations in Stage-B and waiting until they complete before running *op-a*'s continuation.

In practice, designing programs with stages focuses on partitioning tasks into sub-tasks that are self-contained, have considerable local data, and have a logical limit in many ways to process the control flow of object-oriented design.

3.2 Operations

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

The code with 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 to avoid locking operations that suspend threads running operations on a processor. An operation that relinquishes the processor for an event—such as asynchronous I/O, synchronization, or operation completion—instead provides a continuation that is invoked when the event occurs [14].

A continuation consists 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 most costly approach as they save only the state 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 is similar to fibers, which simplifies programming and improves performance [2].

Asynchronous operations provide low-cost parallelism which enables programmers to express and exploit the concurrency within an application. The overhead in the space of invoking operations is close to procedural calls, but it includes allocating and initializing a closure and passing it to a stage. When an operation is completed, it does not require a stack frame, an area to preserve processor state which eliminates much of the overhead. Similarly, returning a closure enables implicit continuation, simplifying expensive 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, a stage can serve as a range of pipeline which requests arrive and the response flows from them. This form of computation is easily supported by representing requests as objects passed between stages. Linear pipelining of this sort is simple and efficient because a stage retains information on completed computations.

However, stages are not constrained to this linear style. Another common programming idiom is bi-directional pipelining which is the synchronous analog of the turn-of-the-table approach that passes subtasks on from one stage to the next. The parent stage eventually suspends work when the request turns its attention to the requests, and resumes the original computation when the subtask produces results. This style requires that an operation be broken into a series of subcomputations, which is where results appear. With explicit continuations, a programmer partitions the computation by hand, although a compiler could easily produce this code which is lost to the well-known continuation-passing style [6, 2]. With implicit continuations, a programmer only needs to indicate where the original computation suspends and waits for the subtask to complete.

A generalization of this style is event-driven programming, which uses finite state automata (FSAs) to control a reactive system [2, 29]. The ESAs are encapsulated and guided by external events, such as network messages and I/O completion, and internal events from other asynchronous stages. An operation's closure contains the ESAs state for a particular server request. The ESAs change state when a child operation completes or external events arrive. These transitions invoke computations associated with edges in the ESAs. Each computation usually locks and specifies the next state in the ESAs.

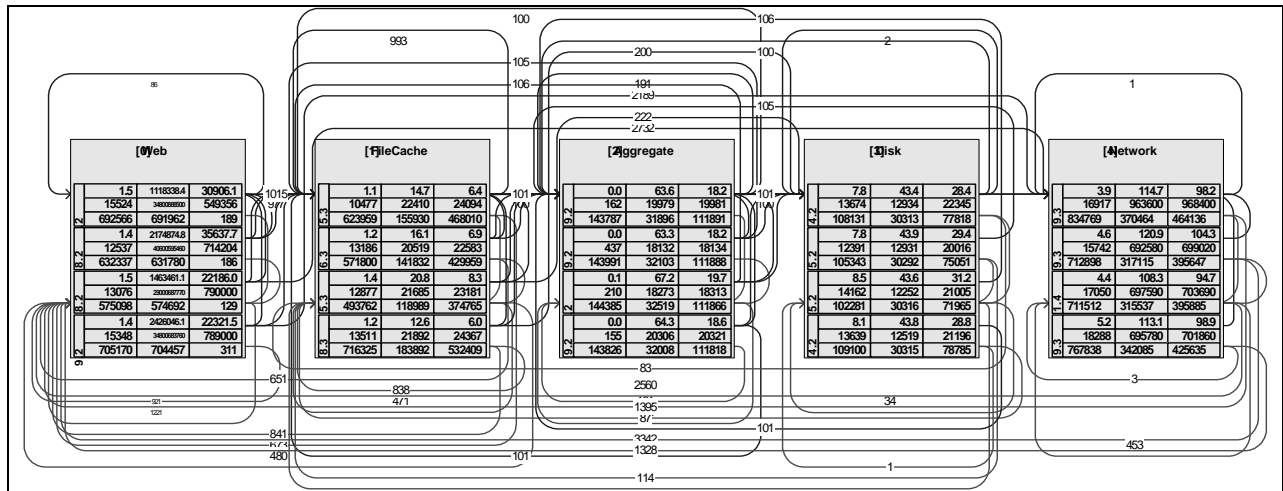


Figure 4 Profile of staged web server (Section 5.1). The performance metrics for each stage are broken down by processor (the system is running on four processors). The first column is the average queue length. The second column contains three metrics on operations at the stage: the quantity, the average 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, we reserve the connection driver control-log stage consists of a fifteen-state FSA describing the process by which HTTP requests are parsed, the file references found in each record, the blocks are read, transmitted, and the connection closed.

Describing the control logic of a FSA opens the possibility of verifying any properties of the entire system, such as deadlock freedom, by applying techniques, such as model checking [22], developed in the verification of communicating FSAs.

3.4 Scheduling Policy Refinements

The initial design of scheduling is autonomous. When a stage is activated, the processor determines which operations to execute and their order. This scheduling freedom allows several refinements to cohort scheduling to reduce the effects of synchronization, particularly in our context, which are useful:

- An *exclusive stage* executes most of its operations in a strictly sequential and completely stage-local manner, not needing synchronization. This type is similar to monitor, except that it interfaces synchronously with the client, delegating computation to the stage, rather than blocking to obtain access to a resource. When this strict serialization does not cause performance bottlenecks, this policy offers a safer, more accessible, and simpler programming model. This approach works well for fine-grained operations that do not require releasing

the stage mutex, can be amortized over a long operation [25].

- A *partitioned stage* divides invocation (based on a key as a parameter), to avoid sharing data among operations running on different processors. For example, consider each stage partitioned using a function of the file number. Each processor maintains its own shared in-memory block cache. Each cache is used only on the 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 can execute concurrently, shared data accesses must be synchronized.

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

It is important to keep in mind that the policies implemented within the more general framework of cohort scheduling. When a stage is activated, the processor executes its outstanding operations, and then another. Nothing in the stage itself requires cohort scheduling. Rather, the programming model and scheduling policy naturally do so. A group of logically related operations can be scheduled to provide the freedom to reorder computations. Cohort scheduling exploits scheduling freedom by consecutively running similar operations.

3.5 Understanding Performance

A compelling advantage of staged models is that the performance of the system is relatively easy to visualize and understand. Each stage is similar to a node in a 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 an overview of system performance and help identify bottlenecks.

3.6 Stage Computation Example

As an example of stage computation, consider the file cache used by the server in Section 1. The file cache is an important component in a server that stores recently accessed disk blocks in memory and maps file identifiers to disk blocks.

The stage file cache consists of three partitioned stages (Figure 5). The cache is logically partitioned across processors so each processor manages a unique subset of files determined by the hash of the file identifier. Alternatively, for large files, the identifier and offset can be hashed together so file's disk blocks are striped across the cache. 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 processors.

The Disk I/O stage reads and writes disk blocks. It 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 scheduling different from a accumulating I/O requests and combining them into larger requests for the operating system.

The Disk I/O stage reads and writes disk blocks. It issues asynchronous system calls to perform these operations and then invokes an operation on the Event Server stage describing the pending I/O. This operation suspends until complete. This stage interfaces with the operating system's asynchronous notification mechanism through a programming model that utilizes separate threads which wait for completion of the operation. Upon completion, this stage asynchronously notifies the I/O Aggregator stage which passes the data to the File Cache stage where the data is recorded and passed to the client.

Figure 4. Performance metrics for a staged model.

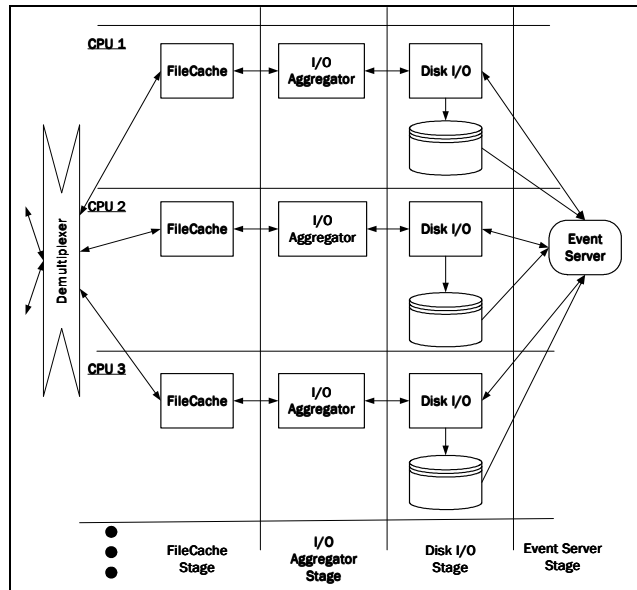


Figure 5. Architecture of staged file cache. Requests for disk blocks are partitioned across processors to avoid sharing the hash table. If a block is not found, it is requested from an I/O aggregator, which combines requests for adjacent blocks and passes them to a disk I/O stage that asynchronously 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 Staged Server Library

The Staged Server Library is a collection of C++ classes that implement staged computation and thread scheduling with a multiprocessor or multiprocessor. This library enables a programmer to define stages, operations, and policies by writing only application-specific code. Moreover, Staged Server implements an aggressive and efficient version of thread scheduling. This section briefly describes the library's primary interfaces.

The library's functionality is partitioned between two principal classes: the Stage class, which provides stage-local storage and mechanisms for collecting and scheduling operations. The second is the Closure class, which encapsulates operations and continuation provides per-invocation state and supports invoking operations and returning results. The fundamental Staged Server system is invoked by creating and initializing a stage.

4.1 Stage Class

The Stage class is a templated base class that applications derive from for various stages. The class provides the functionality for managing operations and scheduling execution on multiple processors.

The closure argument between parent and child continuations and the closure argument between child and grandchild continuations. This process may repeat multiple times with each continuation taking the job of either the closure or the original operation invocation and tries subsequent continuation in practice, treating the method identically and does not distinguish between operation and its continuation.

5 Experimental Evaluation

To evaluate the benefits of co-scheduling and the staged server library we built a prototypical server application. The first—web server—is I/O-bound and consists of responding to HTTP GET requests by retrieving files from disk. The second—publishing server—computes the amount of data transferred relative to a small amount of data to be gained in a database scriptio expensive and memory-intensive.

5.1 I/O-Intensive Server

To compare threads against stages we implemented a web server. The first structure using threads (THWS) is the one-stage server (SSWS). We compare these servers in terms of throughput and compare them against Microsoft Windows's asynchronous I/O operation. The threads were organized in a conventional manner at accepting connections and passing them to 56 work threads, of which perform the server's functionality: reading requests and transmitting responses. This server used the kernel's file cache. The SSWS also processes 56 simultaneously requests. It was organized as a single-stage, network I/O and the disk I/O stages described in Section 3. The parameters were chosen by experimentation in a field of server performance benchmark hardware configuration.

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

Our test configuration consisted of a server and a client. The server was a Compaq Proliant DL580 containing four 700MHz Pentium III-Xeon processors (2MB L2 cache) and 4GB RAM had eight

10000RPM SCSI disks connected to a Smart Array controller. The clients are Dell PowerEdge 6350s each containing four 400MHz Pentium III processors with 1GB RAM. The clients and server were connected by a dedicated Gigabit Ethernet network to a Windows 2000 Server (SP1).

We used the SURGE benchmark, which retrieves web pages whose sized distributions and reference patterns are modeled on actual systems. SURGE measures the ability of a server to process HTTP GET requests to retrieve pages from disk and them back to the client. The benchmark does not attempt to capture the behavior of a server which must handle the type of HTTP requests, execute dynamically content, perform server management, and data. It increases the average configuration with a fixed number of 1000,000 pages (20.1GB) and a stream containing 638,449 requests. A SURGE workload is characterized by User-Equivalent (UE) of which models user accessing the website. We found that we could support 10000 users. All test were run with the workload balanced across the client machines. The reported numbers are for a 5-minute client execution starting with a fresh initialized server.

Figure 1 shows the bandwidth and latency of the threads (THWS) in the one-stage server (SSWS) 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 measured and bandwidth in the second chart contains the average latency perceived by the client to retrieve a page.

Several trends are notable. Under light load, SSWS's performance is approximately 16% lower than THWS. As load increases, SSWS's response time is 3% more requests per minute. The second chart explains this difference. SSWS's latency is higher than THWS's latency under light load by a factor of almost 30. As load increases, SSWS's latency grows only 3 times but THWS's latency increases 45 times. SSWS's

The commercial server Microsoft's IIS outperformed SSWS by 9% and THWS by 22%. Its latency under heavy load was 45% better than other server latency.

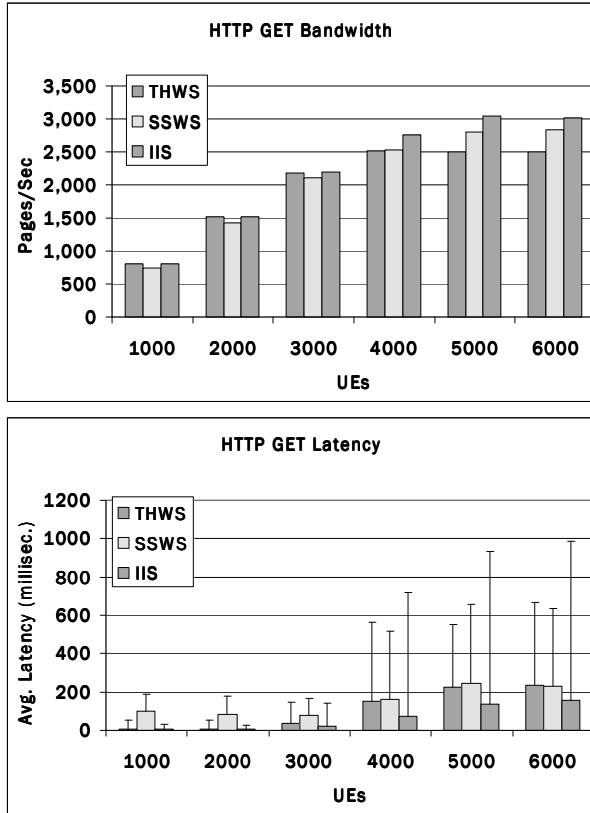


Figure 6. Performance of web servers. These charts show the performance of the threaded server (THWS), StagedServer server (SSWS), and Microsoft’s IIS server (IIS). The first records the number of web pages received by the clients per second. The second records the average latency, as perceived by the client, to retrieve a page. The error bars are the standard deviation of the latency.

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

5.2 Compute-Bound Server

To evaluate the performance of StagedServer on compute-bound application, we also built a publish-subscribe server. The server used an efficient cache-friendly algorithm that events gain an order database subscription. In a subscription conjunction of terms comparing variables against a set of signment of values to generate an event match as subscription and a satisfied value as signment in the event.

Both the threaded (THPS) and StagedServer (SSPS) version of this application shared a common

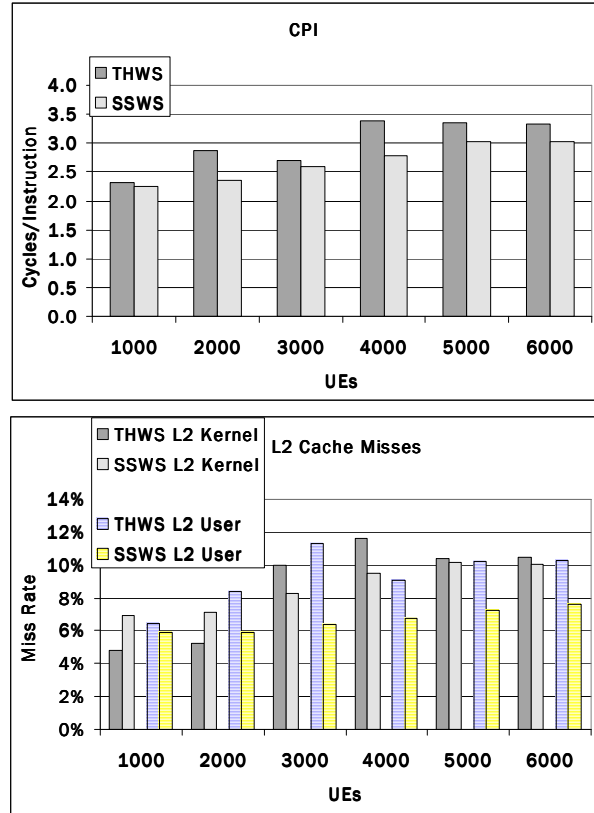


Figure 7. Processor performance of servers. These charts show the processor performance of the threaded (THWS) and StagedServer (SSWS) web server. The first chart shows the cycles per instruction (CPI) and the second shows the rate of L2 cache misses.

publish-subscribe implementation, the only difference between them was the thread structure. The computation benchmark was the Fabret workload, 1000,000 subscriptions and 100,000 event. The platform is also above.

The response time of StagedServer version to events was better than a (Figure 8) with four more client publishing events, the THPS responded an average of 5.7 ms each request. With four clients, SSPS responded an average of 5 ms, and response improved to 4.7 ms with 6 client (31% improvement over the threaded version).

A large measure of this improvement is the improved processor usage (Figure 8) with 6 clients, SSPS averaged 2.0 cycles per instruction (CPI) over the entire benchmark, while THPS averaged 2.6 CPI, a 26% reduction. Over the compute-intensive event matching portion, SSPS averaged 1.7 CPI, while THPS averaged 2.6 CPI, a 33% reduction. A large measure of this improvement is attributable to a greater than 50% reduction in L2 cache misses, from 58% of user-space L2 cache requests (THPS) to 26% of requests (SSPS).

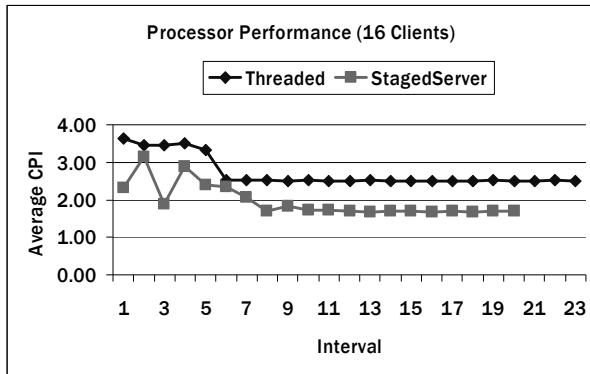
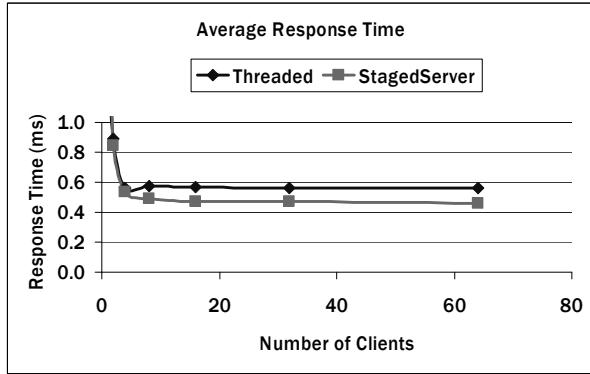


Figure 8. Performance of Publish-Subscribe server. The top chart records the average response time to match publish events against subscriptions. The bottom chart compares average cycles per instruction (CPI) of the thread and StagedServer versions over 25 second intervals. The initial part of each curve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

This application reference data structure (approximately 66.7MB) When matching against subscription Fabret's algorithm although cache-efficient may access large amount of data and the particular locations depend on StagedServer's performance advantage. First, it organizes subscriptions which reduces the number of processor references and hence the possibility of cache misses. Without this locality optimization, SSPS runs at 1.5x the speed of StagedServer. We measured the benefit of scheduling by limiting cohorts of four items reduced SSPS performance by 31% items reduced performance by 7% and event items reduced performance by 16%.

Both optimizations would be beneficial to code structure. The resulting server code isomorphic to StagedServer version with bounded processor performance.

substructure and in each process to accumulate.

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 [6] described the harvest web cache which uses event-driven reactive architecture to invoke computation transitions in state-machine controlled [3]. The system StagedServer uses non-blocking careful avoidance of page faults; and non-blocking non-preemptive scheduling policy [7], [26]. Proposed four-fold characterization of server architectures: multi-process, multi-threaded, single-process event-driven and symmetric multi-process event-driven [26]. The alternative is orthogonal to the task scheduling policy which discusses the locality. Pairwise event-driven programming offers many opportunities for host scheduling since event handlers partition computation into distinct, easily identifiable subcomputations with operation boundaries. Other than ad-hoc event systems offer obvious way to group handlers that belong to the same cohort associated with operation. Section describes staged computation programming model provides programmer with control over computation.

Welsch recently described SED system which is similar to staged computation model [29]. SED, unlike StagedServer does not use explicit cohort scheduling but instead stages as architecture for structuring event-driven servers. High performance results are similar to intensive server applications.

Blackwell used blocked layer processing in providing structural locality [CP/IPack] [8]. He noted that CP/IPack was larger than MIPS R2000 instruction caches, that the protocol stack processes packet completely from the lower protocol layer remains in cache. His solution was to process several packets together at a layer if the modified stack below cache is not reduced processing time. Blackwell's approach blocked matrix computation [9] but its focus was structural locality. Cohort scheduling whose genesis predate Blackwell's more general scheduling policy system architecture which applicable when computation not cleanly partitionable network stack. Moreover, cohort scheduling improve data access instruction locality and reduce synchronization overhead.

Stages similar in some respects to object-based language, it provides local state and operations to manipulate it. However, because subjects are not passive and methods are asynchronous—though synchronous object models exist. Many object-oriented languages such as Java [7] integrate threads, synchronization, and active entities remain thread synchronously. A method that is invoked by a staged computation is asked to perform an operation, given that it can decide to do so and then execute the work. This decoupling of request and response is valuable because it allows a stage to schedule its concurrency to do an efficient scheduling policy such as host scheduling.

Stages similar in some respects to Actor-based communication between autonomous entities. Actors have no internal concurrency and they act over the scheduling. Instead, they resume activity which an Actor responds to by invoking computation stages because of internal concurrency and scheduling autonomy. The unit of work is a host scheduling Actor as in a state data flow or a general computing model [2, 3, 4]. Stages can be viewed as instances of a data flow computation.

Cilk language based on provably efficient scheduling policy. The language is a dot product based but has some characteristics with stage. It is not started computation in an empty. While running computation can spawn off other tasks which return the results by invoking continuation. However, Cilk works stealing scheduling policy to complement host scheduling or under program control. Recent work however has proved the locality of work stealing scheduling algorithm [4].

JAW is a subject-oriented framework for writing servers [8]. It consists of a collection of patterns which can be constructed and particular operating systems selecting appropriate concurrency mechanisms (processes, threads), creating threads, reducing synchronization, infusing scatter-gather, and employing HTTP-specific optimizations. StagedServer simplifies that provides a programming model directly enhance program locality and performance.

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

7 Conclusion

Server commonly structured as a collection of parallel tasks which execute the code necessary to process threads. Processes or event handlers underlie the software architecture of most servers. Unfortunately, this software architecture interacts poorly with modern processors whose performance depends on mechanisms—cache, TLB, and branch predictors—that exploit program locality to bridge the increasing processor-memory performance gap. Servers have little here locality area typically is short and predictable amount of and follow by unrelated threads with this working. Moreover, servers interact frequently with the operating system which can disrupt the working. The process performance of servers is naturally a consequence of their threaded architecture.

As a remedy, we propose host scheduling which increases server locality by consecutively executing related operations from different server requests. Running similar code on a process increases instruction and data locality which hides hardware mechanisms, such as cache and branch predictors. Moreover, this architecture naturally issues operating system requests which reduce system disruption.

This paper also describes the staged computation programming model which supports host scheduling by providing abstraction for grouping late operations and mechanisms through which program can implement host scheduling. This approach has been implemented in StagedServer library. In a series of tests using server and publish-subscribe server, the StagedServer code performed better than threaded code with the level of cache and instruction stalls that better performance can be had.

Acknowledgements

This work is a joint effort and many people have provided valuable insights and feedback. This is incomplete and I apologize for omission. Rick Kicik made important contributions to the design of host scheduling. In a preliminary implementation, Ji Gray has been a seamless support and vocative. The work of Kevin Zatlouk helped write the server and many early experiments. Trishul Chilimbi, Ji Gray, Vinod Grover, Mark Hill, Murali Krishna, Paul Larson, Mimi Marti, Ravi Murthy, Luke McDowell, Scot McFarling, Simon Peyton-Jones, Mike Smith, and Ben Zorn provided many helpful questions and comments. The referees and shepherd, Carl Ellis, provided helpful comments.

References

- [1] Umakanta Gupta, Belloch, and Robert L. Upton. *Localizing the Stealing Problem*. In Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, pages 115-126. San Francisco, CA, 1998.
- [2] Atul Adya, Howard M. Levy, and William R. Douceur. *Cooperating without Managing*. In Proceedings of the 2000 USENIX Security Conference, Monterey, CA, August 2002.
- [3] Guha. *ACTORS: A Model of Concurrent Distributed System Computation*. MIT Press, 1988.
- [4] Anastasia Galimaki, David DeWitt, and David Wood. *DBM: A Model of Process Where Go?*. In Proceedings of the 15th International Conference on Large Databases, Edinburgh, Scotland, Morgan Kaufmann, 1999, pp. 66-77.
- [5] Thomas Anderson. *The Performance Implications of Shared-Memory Multiprocessors*. IEEE Transactions on Parallel and Distributed Systems, 1(6):169-190, 1990.
- [6] Andre W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [7] Gaurang Patil, Peter Druschel, and Jeffrey M. Ousterhout. *Operating System Features for Server Performance*. In Workshop on Internetworking Performance, June 1998.
- [8] Paul Barford, Mark Crovella, and Gregor Vetter. *Generating Workload for Network 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. 51-160.
- [9] Luiz André Barroso, Kourosh Ghahar, and E. Bugnion. *Memory System Characterization of Commercial Servers*. In Proceedings of the 25th Annual International Conference on Computer Architecture, Barcelona, Spain, June 1998.
- [10] Rev. O. J. Blackwell. *Speeding Protocols for Interconnected Networks*. In Proceedings of the 1998 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 1998, pp. 5-9.
- [11] Robert L. Upton, Christoph F. Berger, and Charles R. Leiserson. *Keith Randall and Yuh-Jen Sheng. Efficient Multithreaded Runtime Systems*. Journal of Distributed Computing, 7(1):55-69, 1996.
- [12] Sati Ghahar, Brad R. Richardson, and S. Lavi. *Domain-Specific Language Writing in a Cohesive Environment*. In Proceedings of the 1999 IEEE Transactions on Software Engineering, pp. 17-33, 1999.
- [13] Anwar Chankhunthod, Peter Schwan, and Ken Wilson. *Cache Coherence in a Shared-Memory Multiprocessor*. In Proceedings of the 1999 USENIX Security Conference, San Diego, CA, January 1999.
- [14] Richard D. Bristow, Brian Bershad, and Randal W. Doornik. *Using Continuations to Implement an Efficient Communication Protocol*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, Seattle, WA, October 1999, pp. 212-236.
- [15] Edmund M. Clarke, Ronit Rubinfeld, and Doron D. Dinerstein. *Model Checking*. Cambridge, MA, MIT Press, 1999.
- [16] François Fabre, Harro Jacobsen, François Lirat, and João Pereira. *Implementing a Very Fast Filtering Algorithm*. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, 2000, pp. 15-26.
- [17] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [18] James R. Douglis, Schmidt, JAW, Fram, and Johnson. *High-Performance Servers, a Domain-Specific Framework*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999.
- [19] Frigiano, Crolet, and Supernode. *Partitioning a Server for High Performance*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 319-329.
- [20] Kimberley Keeton, David Patterson, and Qianliang Roger. *Performance Characterization of a Shared-Memory Multiprocessor*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 5-26.
- [21] James R. Douglis, Michael S. Liskov, and John L. Hennessy. *Performance of a Shared-Memory Multiprocessor*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 82-187.
- [22] James R. Douglis, Rajamohanram, and Jakob Rehof. *Behavioral Structure of a Synchronous Programming Language*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 73-79.
- [23] Edward L. Wimmers, and Thomas H. Davenport. *Dataflow Networks*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 192-199.
- [24] Yoshihiro Oyama, Kenji Taura, and Akinori Yonezawa. *Executing Parallel Programs with Synchronization Efficiently*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 82-204.
- [25] Vividus, Peter Druschel, and Willem Zwane. *Efficient Portable Web Server*. In Proceedings of the 1999 USENIX Security Conference, Monterey, CA, August 1999, pp. 99-212.
- [26] David Patterson and John L. Hennessy. *Quantitative Approach to Performance Analysis*. In Proceedings of the 1999 USENIX Security Conference, Monterey, CA, August 1999, pp. 69-183.
- [27] Sharad Aravamudan, Richard Sites, and David D. Clark. *Performance of a Shared-Memory Multiprocessor*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 69-183.
- [28] Ma Wels, David D. Clark, and R. S. Sutton. *Scalable Server Architectures*. In Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, San Francisco, CA, October 1999, pp. 30-243.
- [29] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.