Asynchronous Delegation and its Applications

by

George Dill
BSc (Purdue University, West Lafayette, IN) 2008

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

ACKNOWLEDGMENTS

The thesis has been completed... (INSERT YOUR TEXTS)

YOUR INITIAL

PREFACE

TABLE OF CONTENTS

CHAPTER **PAGE**

1 **BACKGROUND AND MOTIVATION** 1
- 1.1 Synchronization 1
- 1.2 Delegation 2

2 **ASYNCHRONOUS DELEGATION** 6
- 2.0.1 General 6
- 2.1 Asynchronous Dedicated Delegation 8
- 2.1.1 Client and Server Production Rates 10
- 2.1.2 More Concurrency with Request Lines 12
- 2.1.3 Hurry Up and Wait - The Pending Request Queue 13
- 2.2 Asynchronous Flat Delegation 18
- 2.2.1 Scheduling Strategies for Asynchronous Flat Delegation 19
- 2.2.1.1 Pending Request Queue 21
- 2.2.1.2 Direct Write Serve When Blocked 22
- 2.2.1.3 Direct Write Fast Path, Greedy Client, Upper Bound, and Doorbell 23
- 2.3 Ordering Guarantees 24

3 **EXPERIMENTAL EVALUATION** 26
- 3.1 Fetch and Add 26
- 3.1.1 Performance Under Contention 28
- 3.1.2 Memory Subsystem 29

4 **CONCLUSIONS** 32

CITED LITERATURE 33

LIST OF TABLES

TABLE		PAGE
I	Excerpt of the asynchronous delegation API	7

LIST OF FIGURES

FIGURE		PAGE
1	A delegation system with 3 clients and 3 servers. The foreground client makes a request to all three servers.	3
2	A delegation system with 1 client and 3 servers. The foreground client makes a request to all three servers. Requests are written to a pending request buffer which is periodically flushed out to the request line.	8
3	Throughput in MOPS for fetch and add vs fetch and sqrt. While server speed is faster than client speed, peak throughput will require more servers when the delegated function is slower.	10
4	Throughput in MOPS for a 56 thread system by number of request lines.	12
5	Throughput in MOPS for a 56 thread system by number of servers with a 32 length pending request queue and direct request line writes.	14
6	Throughput in MOPS for a 56 thread system by number of servers by pending request queue length. The remainder of threads are clients.	15
7	Latency, Server Saturation, Response Readiness, and Failed Queue Pops by queue length.	16
8	A flat delegation system with 3 client-server threads. The foreground client makes a request to itself and the background client.	18
9	Flat delegation throughput by scheduling policy.	20
10	56 server-client pair comparison of average request latency, response line availability, and comparative throughput. Greedy omitted for scale.	21
11	28 server-client pair comparison of average request latency, response line availability, and comparative throughput. Greedy omitted for scale.	22
12	The server may execute requests out of order when multiple requests lines are used by one client. (1.) The client issues 3 requests before the server reaches its section of the request line array. (2.) The server handles all requests in this clients section before the client writes a fourth request. (3.) The client writes its next request into the next available line, after request 4 is written the next line is the first one. The requests are now executed out of order.	24
13	Throughput in MOPS by number of 64Byte variables. Higher is better.	27
14	Throughput by Shared Variables by Number of Dedicated Servers	30
15	Delegation loads from RAM	31

LIST OF ABBREVIATIONS

AMS	American Mathematical Society
CPU	Central Processing Unit
CTAN	Comprehensive TEX Archive Network
FFWD	Fast, Flyweight Delegation
FIFO	First In First Out
KB	2^{10} Bytes
MOPS	Million Operations per Second
NUMA	Non Uniform Memory Access
TUG	TEX Users Group
UIC	University of Illinois at Chicago
UICTHESI	Thesis formatting system for use at UIC.

SUMMARY

Synchronization by delegation has been shown to increase total throughput in highly parallel systems over coarse grained locking, [?] but the latency inherent in passing messages between cores introduces a bottleneck in overall throughput. To mitigate the effects of this bottleneck we introduce parallelism in message passing by enabling asynchronous delegation calls.

We present an asynchronous design for both dedicated and flat delegation strategies. In dedicated delegation hardware threads act exclusively as a client or server as opposed to flat delegation where hardware threads share duty as both client and server.

This work is based upon *Gepard* which provides parallelism by using fibers, a user space threading library. Our asynchronous approach removes the memory and computation overhead of switching between fibers, freeing cache resources and processor cycles. Even more concurrency in message passing is added to the *Gepard* server design by increasing the number of requests lines per server from 4 to 16. The result is a throughput increase of up to 400 MOPS on our test bench.

We compare the designs and throughput of asynchronous delegation to that of Gepard [?], fine grained locks, and atomic operations on a fetch and add microbenchmark. We find that dedicated asynchronous delegation outperforms all other synchronization schemes tested when the delegated data structure remains in the server's cache.

Flat asynchronous delegation performs comparatively well when delegated data structures are small and competing approaches experience high contention rates for locks or exclusive mode. Flat delegation performs comparably to fine grained locking approaches for very large shared data structures where DRAM access latency dominates performance.

CHAPTER 1

BACKGROUND AND MOTIVATION

1.1 Synchronization

Traditionally, parallel programs have operated under a model where multiple threads perform operations on the same data structure. Threads operating on shared memory encounter a data race when they are simultaneously reading and modifying the same memory address. This condition results in a non-deterministic outcome of the program and is notoriously difficult to debug. Synchronization of shared memory has been traditionally achieved using atomic operations, when feasible, or a variety of mutual exclusion locking approaches.

For access to individual words, locked atomic instructions provide a guarantee that a write will be seen consistently across all threads. The guarantee can be made by locking the system bus, but is more often achieved through the processor's cache coherency policy [**?**]. The locked operations are slower than their standard counterparts; for example the locked compare and exchange instruction, LOCK CMPXCHG, on Intel Skylake takes 18 cycles while a CMPXCHG instruction takes 6 [**?**].

The compare and exchange instruction can be used to implement a mutual exclusion (mutex) lock. Upon a acquiring a lock though some type of locked atomic instruction, a thread can proceed to perform a more complicated critical section using non-atomic instructions, and then release the lock. This programming pattern effectively serializes accesses to shared memory and eliminates the data race. However when a lock is contended, threads may spend an inordinate amount of time waiting to acquire the lock rather than performing useful calculations.

1.2 Delegation

Delegation, as described in [**?**], serializes accesses to shared memory by granting exclusive control of a data structure to a single thread called a *server*. *client* threads requiring access to the data structure will delegate the operation to the *server* through a request. The requests are handled one-at-a-time by the *server*; serializing the memory access and eliminating the race condition.

Specifically, in *FFWD* style delegation, servers receive requests to perform an operation on their memory from *clients* via a 64 Byte struct containing a function pointer, up to 6 arguments, and a status flag. Similarly responses are communicated via a 16 Byte struct containing a return value and a status flag.

Every server-client pair has at least one dedicated request line and one dedicated response line. Figure 1 shows a system with 3 clients and 3 servers. The client in the foreground makes a request to its allocated request line on all three servers. The server performs an operation on its delegated data structure, and writes the response to the response line allocated to that specific client. Since each request and response line has only one writer, there is no data race nor any cache line contention.

Each delegation *server* iterates through an array of requests. When a new request is encountered the server loads the parameter values included in the request to the appropriate registers and then calls the function pointer. The server stores the return value from the function and

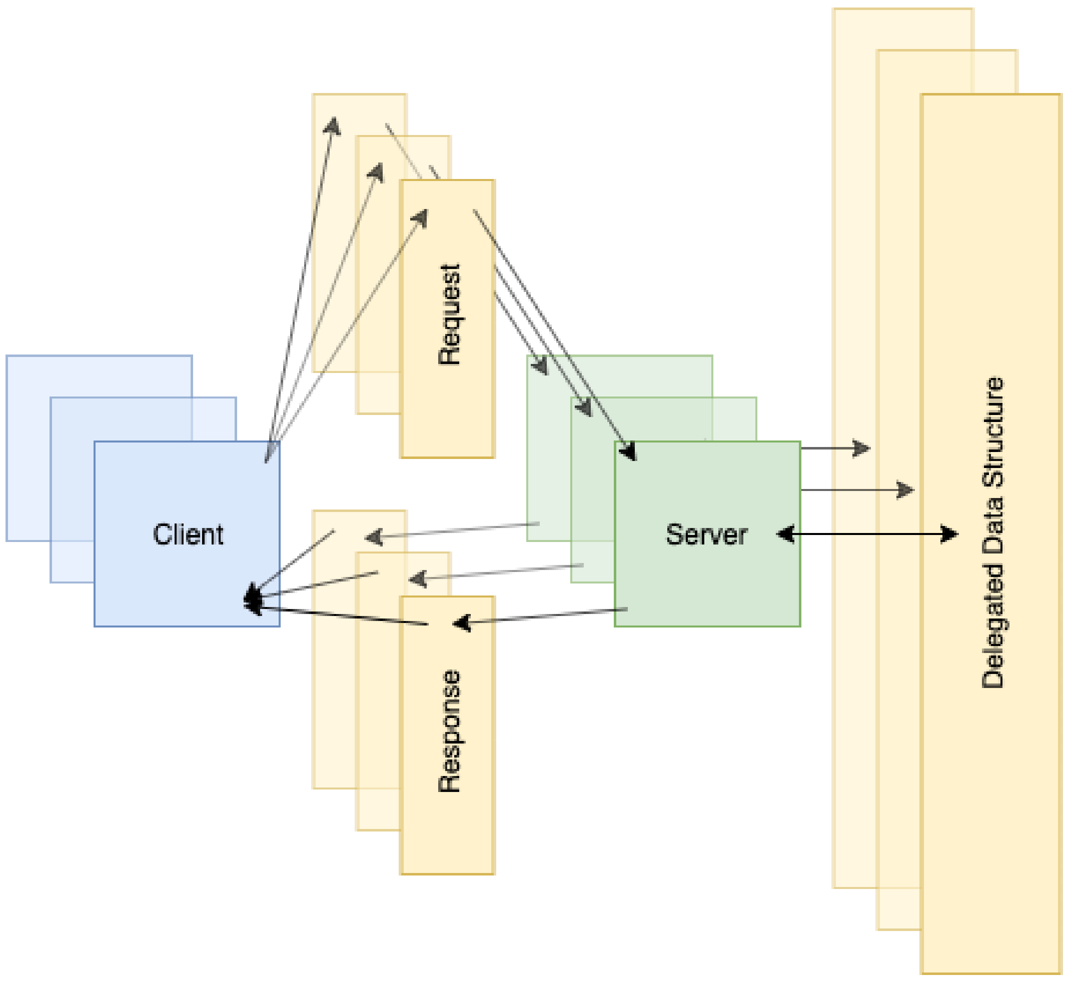

Figure 1. A delegation system with 3 clients and 3 servers. The foreground client makes a request to all three servers.

the flag variable into an array of responses. As a single thread, the *server* can executes requests sequentially, guaranteeing atomicity of delegated functions.

Programs using the delegation library typically initialize the data structure to delegate, launch the dedicated servers, and then launch threads running the client function through a POSIX threads like interface.

An advantage of delegation is spatial locality of memory. A block of memory accessed exclusively by a delegation server is never shared with another thread. When the delegated memory block is sufficiently small, the entire working memory may fit within a server's higher levels of cache and remain resident for the duration of the program. In contrast, a system with multiple physical cores accessing the same size memory block will share cache lines, greatly reducing the likelihood of a higher level cache hit.

From the client perspective, a drawback of delegation is the latency from request issuance to response. In *FFWD*, a synchronous delegation system, clients issue a request to the server's request line and then poll the respective response line until the request is returned. The time to complete a single delegated operation includes the time to write to the server, perform the function, and then receive the response.

Gepard introduces concurrency in delegation operations while maintaining a synchronous appearance to the programmer by enabling a thread to switch to productive work through the use of fibers. Based upon *libfiber* [**?**], fibers are light-weight, cooperatively scheduled user mode threads. Gepard clients write a delegation request to the server then make a rapid context switch to another working client fiber. After some time the client fiber will be reactivated, read its response, and continue execution. The major advantage of *Gepard* is that it enables a single thread to engage with multiple servers concurrently; increasing throughput despite constant individual request latency.

In *Gepard* and *FFWD* the client application blocks after issuing a request until a response is received from the server. *Gepard* hides the latency of the request by switching fibers during this waiting period. However if we can tolerate an asynchronous programming model we can expose greater concurrency without the overhead of switching user space fibers.

CHAPTER 2

ASYNCHRONOUS DELEGATION

2.0.1 General

The primary contribution of Asynchronous delegation is the decoupling of the delegation call from its response. While *Gepard* and *FFWD* make a call to **delegate(s, retval, f, args)** and block until a value is returned, asynchronous delegation makes a call to **Delegate_Async(s, cb, f, args)**, which, for most invocations, is non-blocking, and continue to generate more requests. Actions required upon the return of the delegation request can be executed at a later time through the use of the callback function, cb.

An excerpt of the asynchronous delegation API is shown in 2.0.1.

The new API allows asynchronous delegation to completely remove *libfiber*, the enabler of client side concurrency in *Gepard*, and its memory and computation overhead. While *Gepard* keeps track of a 1KB run time stack for each fiber to maintain the state of a blocked delegation call, asynchronous delegation keeps track of an 8B function pointer, if necessary, to complete a delegation action. Peak throughput for most *Gepard* benchmarks corresponds with 64 fibers per thread, meaning *Gepard* makes frequent accesses to 64KB just to switch context between fibers, and consumes computational resources to make the switch.

Function	Description
Launch_Servers(n)	Starts the specified number of server threads, allocates and initializes the request and response lines.
Client_Thread_Create(f, arg)	Allocates and initializes a pending request queue for every server as thread local variable. Launches an OS thread to run function f with argument arg.
Delegate_Async(s, cb, f, args...)	Generates a delegation request to server s with function f and arguments args. Calls cb with the return value from f.
Delegate_Async_Barrier()	Places requests from a delegated thread's queue and polls server responses until all requests have been served.

TABLE I

Excerpt of the asynchronous delegation API

In this chapter we will discuss the asynchronous delegation with one or more dedicated delegation server threads. Later, we will show how asynchronous operation can be applied to flat delegation, a more recent, serverless design.

2.1 Asynchronous Dedicated Delegation

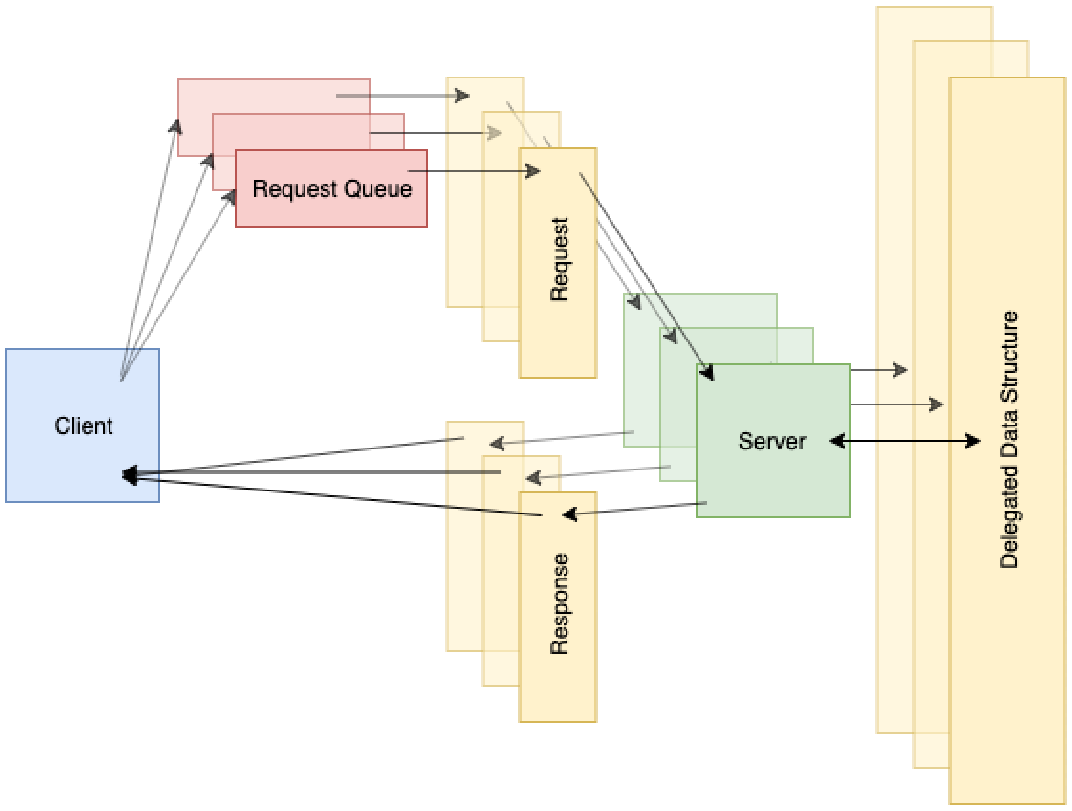

Figure 2. A delegation system with 1 client and 3 servers. The foreground client makes a request to all three servers. Requests are written to a pending request buffer which is periodically flushed out to the request line.

Figure 2 demonstrates a dedicated delegation system with 1 client and 3 servers. In general, the client generates requests and places them in the fixed-length request queue for the required server. When any of the queues fills, the client will suspend request generation and begin the process of writing out requests to servers. The client polls its response lines one by one, executing the callback function if provided. If a response was present a request is popped from the request queue and placed in the request line corresponding with the recently served response. After all response-request lines have been handled the client returns to generating requests.

Specifically, to use dedicated delegation as shown in Figure 2, the user must first initialize a number of delegation servers using **Launch_Servers**. Running on separate OS threads, the servers begin sequentially polling their request lines. The user then launches OS threads with the application code by calling **Client_Thread_Create**. The thread launched by **Client_Thread_Create** first initializes an empty queue of pending requests for each running server on the NUMA node which the client thread is assigned. The queue is implemented as a fixed size circular buffer to take advantage of the prefetcher by making reads of consecutive memory addresses. After queue initialization the thread invokes the client function.

Within the client function the programmer uses **Delegate_Async** to delegate the request. Typically **Delegate_Async** will enqueue the request locally and then return to the client function. However, when any of the pending request queues is filled, the client works through its entire list of responses. When a response is ready, the client invokes the callback with the corresponding return value, and then writes a new request to the corresponding request line from the pending request queue.

At any time the user can call **Delegate_Async_Barrier** to ensure that all outstanding requests are served before moving on. **Delegate_Async_Barrier** is always called after the return of the client function and before joining the client thread.

2.1.1 Client and Server Production Rates

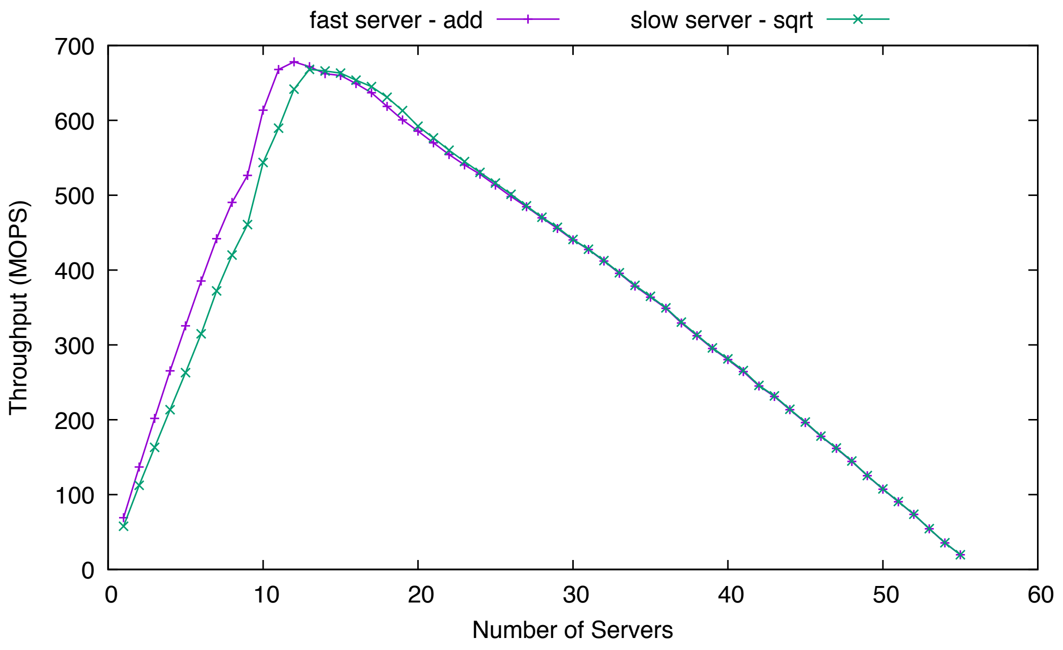

Figure 3. Throughput in MOPS for fetch and add vs fetch and sqrt. While server speed is faster than client speed, peak throughput will require more servers when the delegated function is slower.

A feature of dedicated delegation is the ability to select the number of clients and servers operating in the system. This is a course grained way for the programmer to balance the expected request production rate of a client with the expected consumption rate of the server. In general, the programmer should increase the number of servers for more time consuming delegated functions.

To quantify this rule of thumb, consider the throughput of an idealized system following 2.1.1. Where $\mathrm{Thput}_{\mathrm{system}}$ is the throughput of the entire system in MOPS, $\mathrm{Thput}_{\mathrm{component}}$ is the throughput of an individual component in MOPS, and $\mathrm{N}_{\mathrm{component}}$ is the number of that component in the system. As $Thput_{server}$ decreases, N_{server} must increase to compensate until it reaches equilibrium with $N_{client} * Thput_{client}$.

$$Thput_{system} = \min(N_{server} * Thput_{server}, N_{client} * Thput_{client})$$

Consider Figure 3, an experiment where we compare the throughput by number of servers of a faster (addition) and slower (sqrt) delegated function. For the faster function a client selects a server at random and delegates a function to increment a counter on that server. For the slower delegated function the server delegates an increment and a square root calculation. In the regions left of peak throughput shown in Figure 3 the system is bound by server capacity. The upward slope corresponds to the consumption rate of an individual server, or $\mathrm{Thput}_{\mathrm{server}}$. Conversely, in the region to the right of peak throughput the system is bound by client production. The downward slope corresponds to the $\mathrm{Thput}_{\mathrm{client}}$.

Figure 3 shows 2.1.1 to be a good model. Consumption rate slows as we slow the server performs a more time consuming delegated function. The slower $Thput_{server}$ requires more N_{server} to balance the total throughput of the clients. The peak throughput, or equilibrium point, requires more servers.

2.1.2 More Concurrency with Request Lines

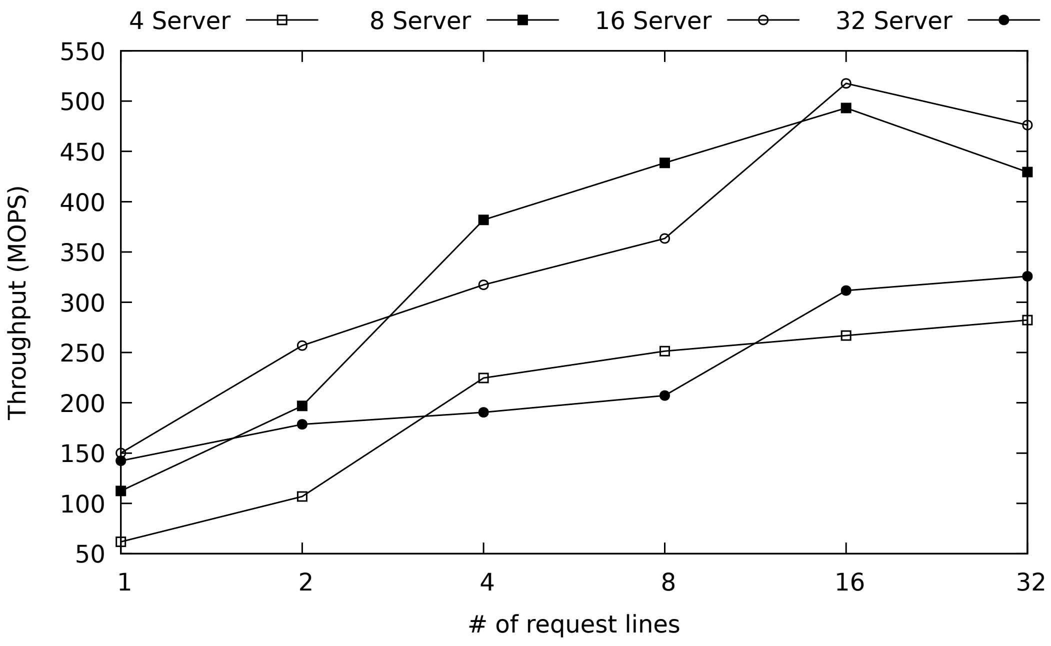

Figure 4. Throughput in MOPS for a 56 thread system by number of request lines.

Beyond using more servers for concurrency, we can adjust the number of requests that can be sent to any one server at a given time by increasing the number of request lines. a server polls while in operation.

Figure 4 shows the measured system throughput for the same benchmark described as the fast server in **??** with throughput plotted against the number of request lines per server-client pair. Figure 4 shows that increasing the number of request line generally increases the throughput until about 16 request lines when performance degrades. Due to this observation we use 16 request lines for the rest of our experiments.

2.1.3 Hurry Up and Wait - The Pending Request Queue

A feature of the asynchronous API is the ability to cache requests until the client is ready to write them out to the server. Running client application code for contiguous periods of time, as opposed to breaking to write requests and read responses, should provide some benefit of temporal locality. The benefit should also apply when writing requests to the server as a batch rather than individually.

The performance advantage of the pending request queue is evident in Figure 5. Figure 5 shows the same fetch and add experiment as **??**, however this time the throughput of an asynchronous dedicated delegation configuration with a 32-length pending request queue is plotted against the throughput of a system configured with direct writes to a server's request line. The client keeps track of the last request line written. When a call to **Delegate_Async()** is made the client attempts a write to the next request line. If the line is unavailable **Del-**

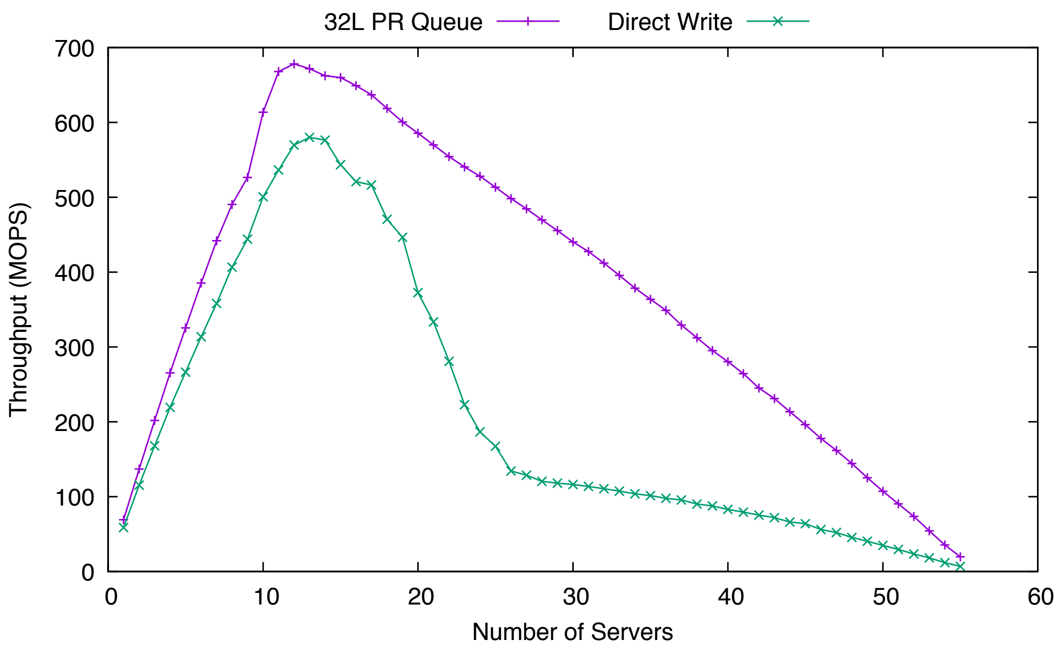

Figure 5. Throughput in MOPS for a 56 thread system by number of servers with a 32 length pending request queue and direct request line writes.

egate_Async() tries the write repeatedly until it is successful. Although direct request line write shows a respectable peak throughput, its rapid degradation in performance is undesirable.

We move forward with implementing the request reserve as an array of pending request queues. For each client, a FIFO queue of fixed length, implemented as a circular buffer, is allocated for each server. Each call to **Delegate_Async()** attempts to push a request onto the queue. When a push request fails, **Delegate_Async()** iterates through its response and request lines, calling the callback on responses if provided and writing requests from the pending request queue to the server if available.

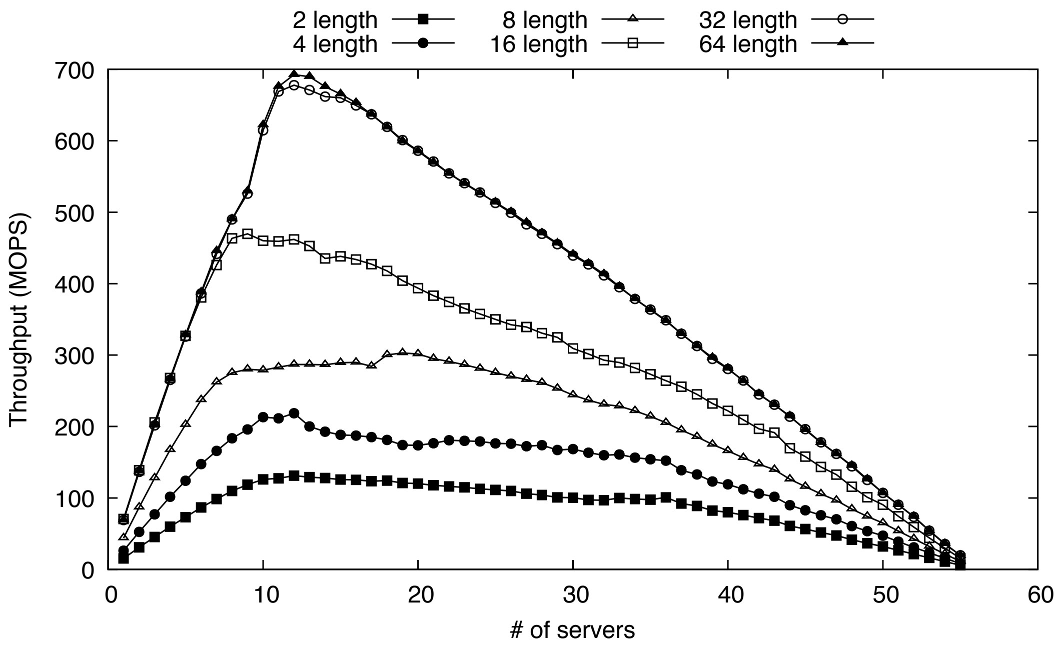

Figure 6. Throughput in MOPS for a 56 thread system by number of servers by pending request queue length. The remainder of threads are clients.

From Figure 6 we see that the length of the pending request queue has a clear impact on overall throughput. Figure 6 shows our familiar benchmark plotted with various pending request queue lengths. The marginal return on throughput for longer queue lengths diminishes

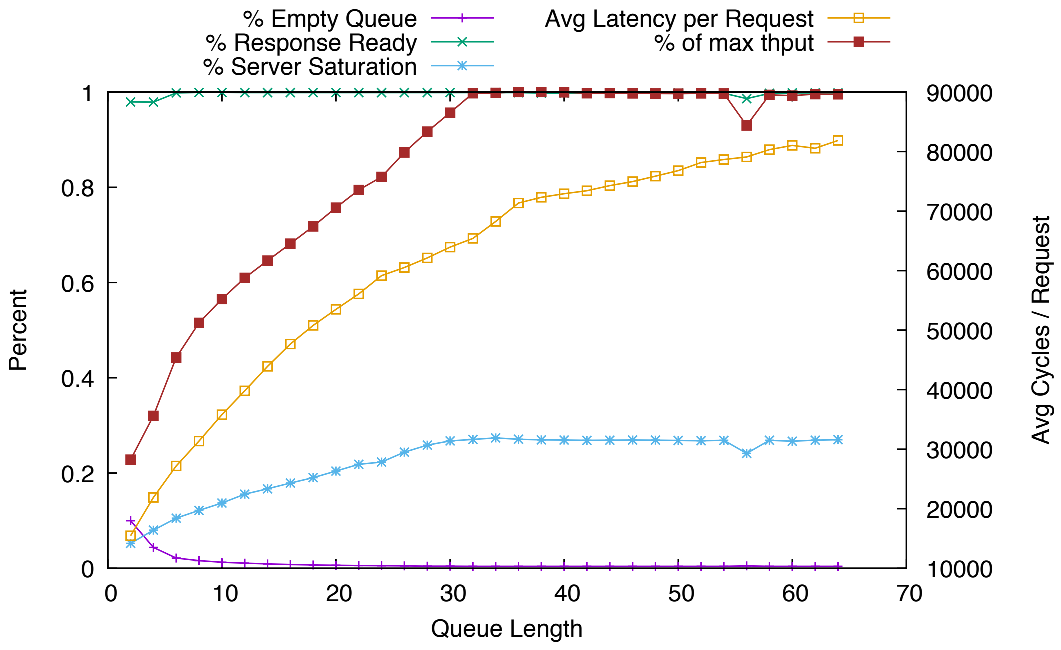

Figure 7. Latency, Server Saturation, Response Readiness, and Failed Queue Pops by queue length.

after length 32, or $2 * N_{requestlines}$. For more insight into this, we reference Figure 7, which shows the saturation of the server, or rate of ready requests to request line reads against the queue length for a system running the same benchmark with 16 servers and 40 clients. Server saturation increases with the extension of the pending request queue, meaning the servers are simply doing more work with fewer reads.

Figure 7 also provides insight into the relationship between the pending request queue length and overall throughput shown in Figure 6. **Delegate_Async()** blocks to read responses and write requests whenever a push to any of its request queues fails. Ideally, **Delegate_Async()** should block at a frequency slow enough that servers have had time to execute the outstanding responses and cache many requests to write concurrently but fast enough that responses do not age on the response line.

The probability that **Delegate_Async()** will block with slower frequency increases with the length of the request queue. The probability, P, that a system with S queues of length L will block after L total requests follows $P = \frac{S}{L^S}$. Clearly the probability of stopping after L total requests drops rapidly as the queue length increases. Intuitively, this also provides the opportunity for requests to be cached across a range of servers. When the queue is sufficiently long and there is diversity in requests, we have a good probability that each queue will hold requests to fill all the request lines when **Delegate_Async()** blocks.

Figure 7 confirms all of the above desirable traits of the queue length. As the queue length increases, the proportion of failed queue pop operations plummets. The increased period between response line polls also results in higher response availability. Server saturation also rises as the queue length increases. After increasing pending request queue length to double the number of request lines, however, the benefits of the longer queue begin to erode. This is shown by the 64 length queue in Figure 6. When the queue is too long it acts like a reservoir, impounding requests for a longer period of time without impacting the throughput of the system. This is shown in Figure 7 by the increase in individual request latency as the queue length increases and increase in throughput stagnates.

As a rule of thumb, we set the pending request queue to double the amount of request lines per client - server pair. This configuration provides high throughput while leaving cache available for delegated data structures.

2.2 Asynchronous Flat Delegation

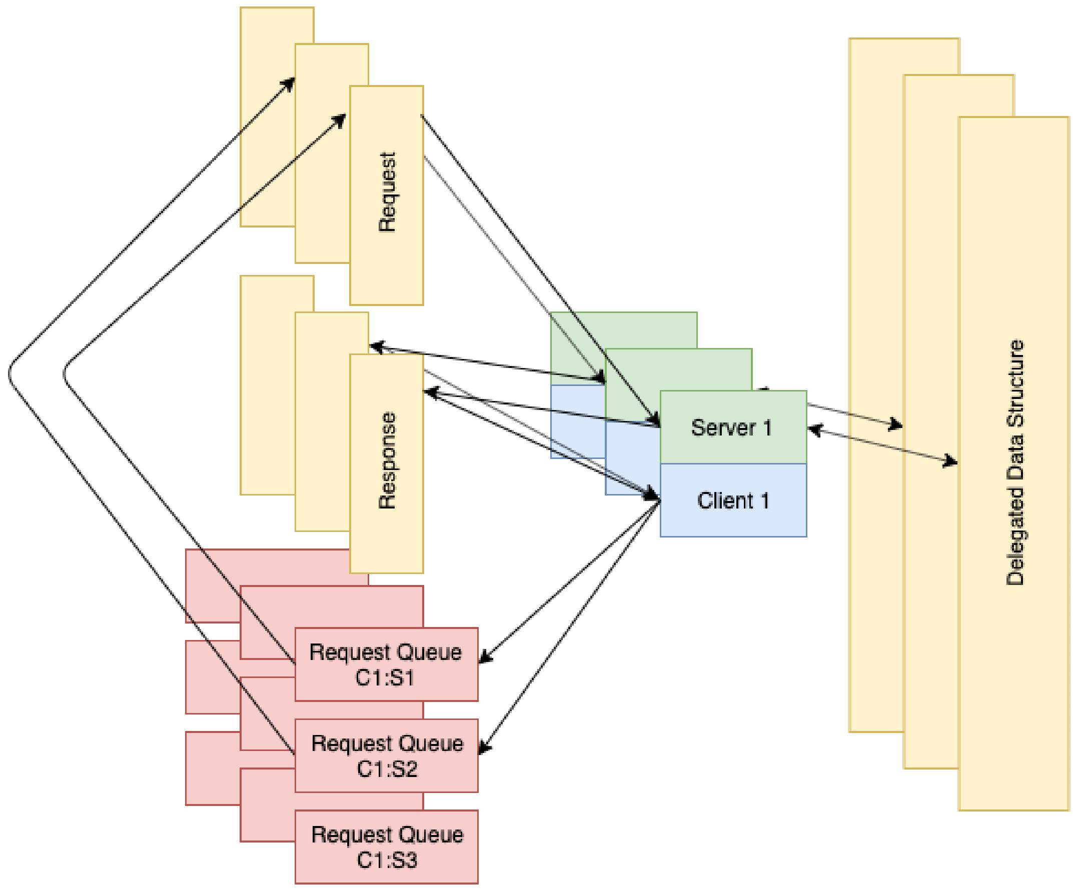

Figure 8. A flat delegation system with 3 client-server threads. The foreground client makes a request to itself and the background client.

Flat delegation combines the client and server roles of dedicated delegation into a single thread. The goal is to simplify delegation programming by finding a natural equilibrium between client and server rather than the user specified client to server ratio of dedicated delegation. Figure 8 sketches a sample flat delegation system with three OS threads splitting duty as client and server. Functions performing client and server duty are called from the same thread represented by the split box in the center of the figure. In Figure 8 the client generates requests and writes them out to the request line of the desired server, which may be the server to be invoked later on the same thread. Depending on the scheduling policy, the server function will be called to handle the requests on its request line.

Since flat delegation does not launch dedicated server threads there is no call to **Launch_Servers**. Besides this difference the API remains the same as dedicated delegation.

Although the API for asynchronous flat delegation remains the same, within **Delegate_Async()** the client must determine when to invoke the server function in addition to caching and writing out requests. Flat delegation servers run periodically as determined by the scheduling policy instead of continuously as they do in the dedicated approach. Because of this key difference, we continued to experiment with writing requests to servers directly as described in **??** which enabled us to try a handful of strategies for invoking the server function.

2.2.1 Scheduling Strategies for Asynchronous Flat Delegation

Figure 9 summarizes the throughput for the asynchronous flat delegation scheduling strategies tested on the same fetch and add benchmark as **??**. The independent variable, however represents the total number of threads running, e.g. 20 means there are 20 threads running as

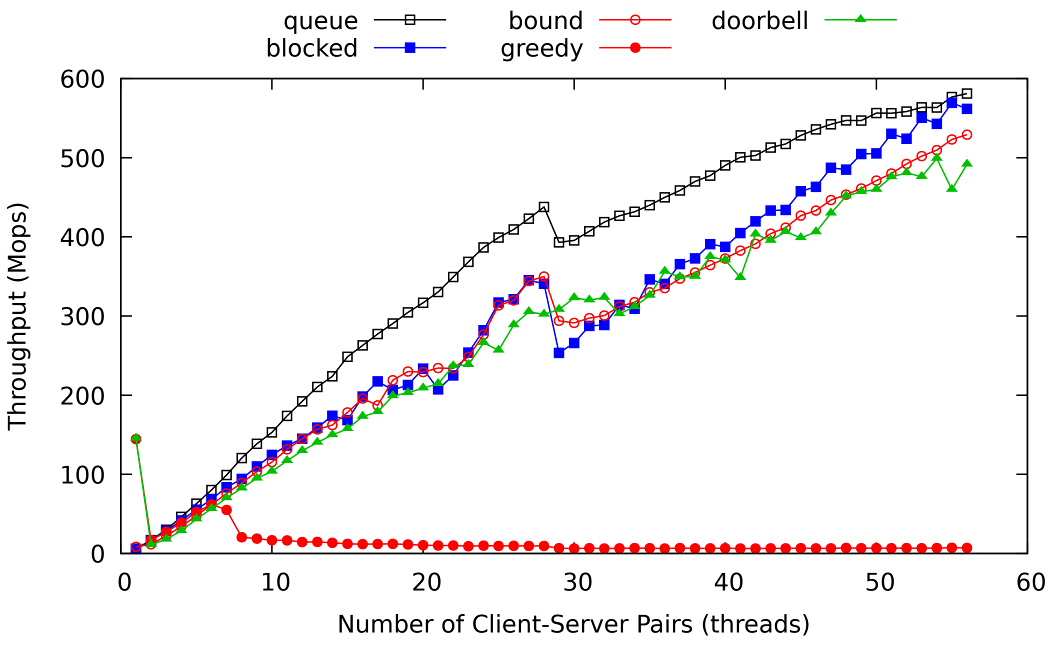

Figure 9. Flat delegation throughput by scheduling policy.

both client and server. The parameters for each are described in the following paragraphs. The server function is the same server function as Asynchronous Dedicated Delegation and *Gepard* except it iterates through its request lines once and then returns.

Caching requests for batch write continues to be the highest throughput (*queue*), however direct request line write (*blocked*) shows comparable performance for higher numbers of threads.

Figure 10 and Figure 11 show the average latency in clock cycles and proportion response ready when polled by delegation strategy. The Direct Write Serve When Blocked strategy shows minimum average request latency.

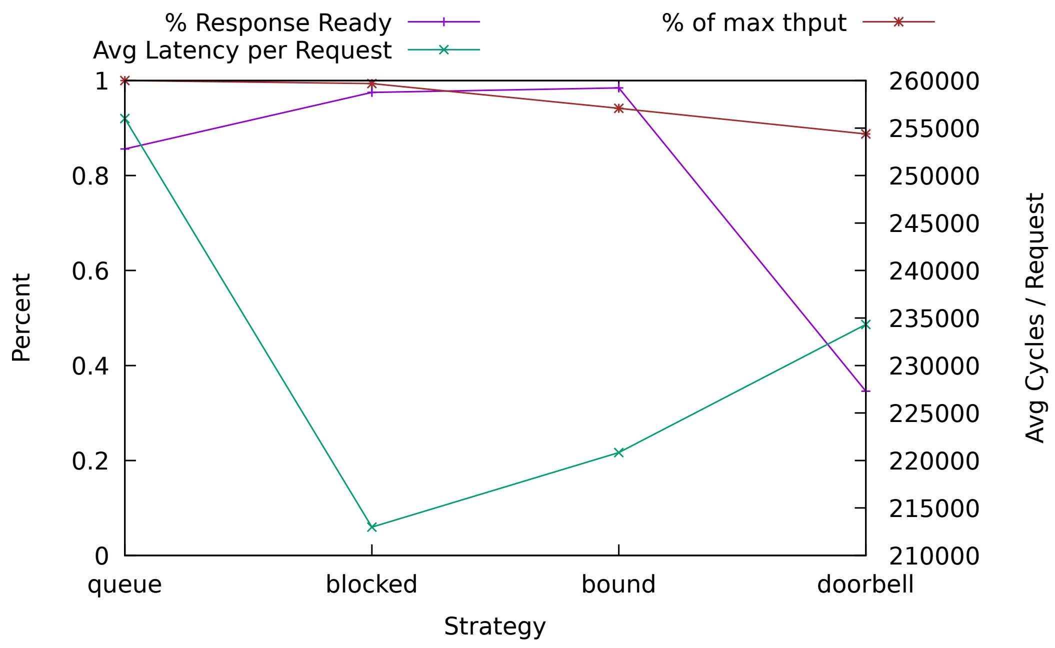

Figure 10. 56 server-client pair comparison of average request latency, response line availability, and comparative throughput. Greedy omitted for scale.

The following subsections provide more detail on the request writing and server invocation strategies.

2.2.1.1 Pending Request Queue

Shown as *queue* in Figure 9 pending request queue is the same as direct delegation described in **??** but with the server function invoked prior to writing and reading request lines. This approach outperformed all others in our trial, and is the approach tested in **??**.

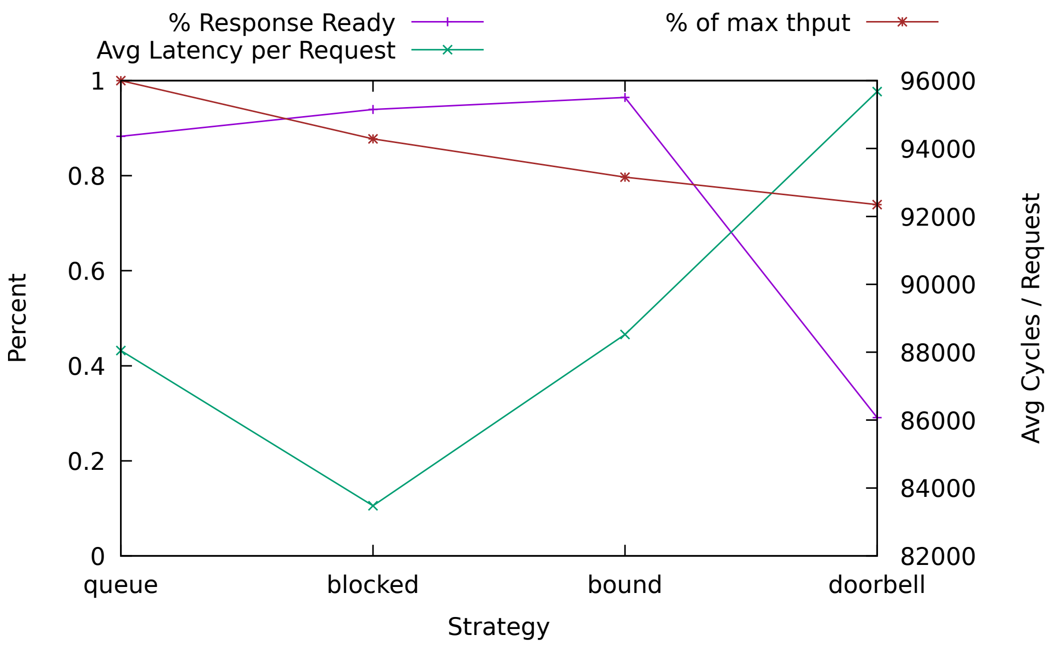

Figure 11. 28 server-client pair comparison of average request latency, response line availability, and comparative throughput. Greedy omitted for scale.

2.2.1.2 Direct Write Serve When Blocked

Perhaps the simplest strategy for invoking the server function is upon a failed request issuance. After all, if the client is blocked waiting for a server to become available, the most useful thing it can do is invoke the server to handle others' requests. As shown in **??** *blocked* shows comparable results to *queue* for larger numbers of threads. *Blocked* cuts down on delegation memory footprint because it eliminates the pending request queues. For larger systems, or systems with smaller amounts of on core cache *blocked* may provide a way to better utilize the on core cache while maintaining higher throughput.

The *blocked* strategy exhibits a desirable property shown in Figure 10 and Figure 11. The latency for any individual request, from generation to execution of the callback function is 16.8% lower for *blocked* than *queue*.

2.2.1.3 Direct Write Fast Path, Greedy Client, Upper Bound, and Doorbell

A proposed optimization to the serve when blocked policy was to fast track requests to a server on the same thread as the client by invoking the delegated function directly. The unintended impact of this optimization was the *greedy* client problem, which is caused by victim threads attempting to write requests to the full request lines of the greedy client. The victim threads operate in server mode until the request line becomes available. However, the request line never becomes available because the greedy client's requests continue to be served by the victim threads. The greedy client problem was avoided in *blocked* strategy because the client was guaranteed to invoke the server function when writes to the request lines of its own server blocked.

To break the greedy client we experiment with an upper *bound* on the client by invoking the server after a fixed number of calls to **Delegate_Async()**. We also implement the *doorbell* strategy. When a client blocks on an unavailable request line it "rings the doorbell" of the required thread by writing a 1 to its doorbell variable. Clients check their doorbell during each call to **Delegate_Async** and invoke the server, resetting the doorbell to zero after a run of the server loop.

2.3 Ordering Guarantees

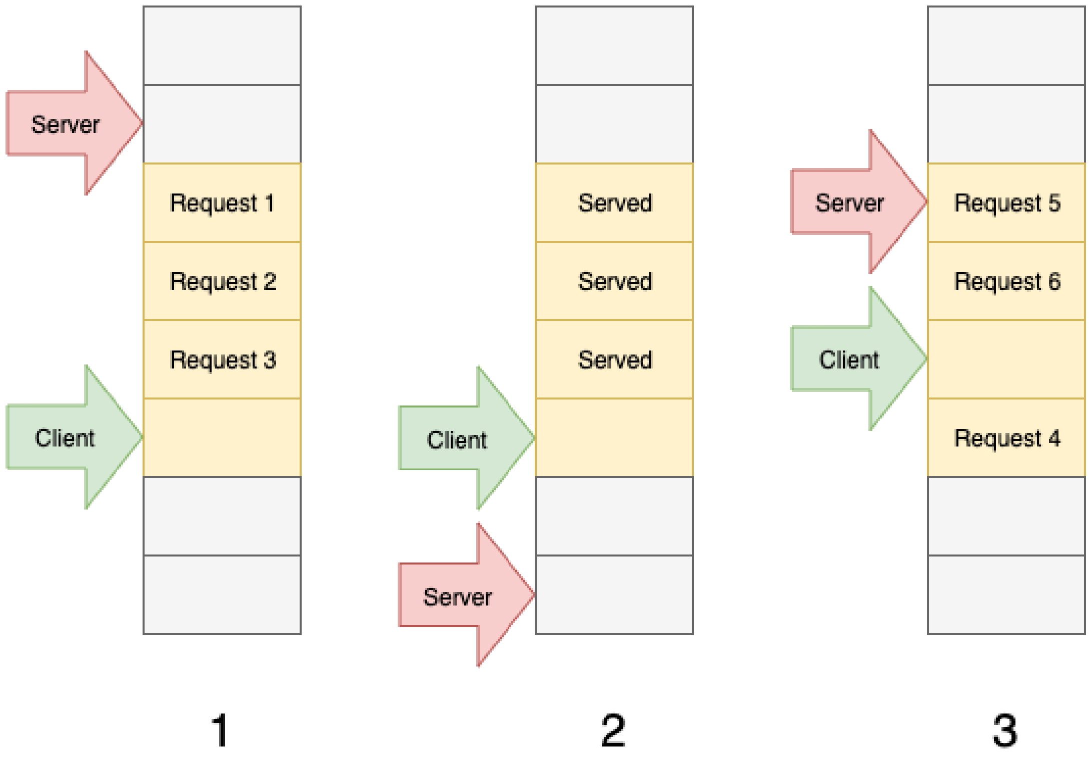

Figure 12. The server may execute requests out of order when multiple requests lines are used by one client. (1.) The client issues 3 requests before the server reaches its section of the request line array. (2.) The server handles all requests in this clients section before the client writes a fourth request. (3.) The client writes its next request into the next available line, after request 4 is written the next line is the first one. The requests are now executed out of order.

The asynchronous API provides no guarantee to the ordering of delegated functions. Delegated functions may be executed out of generation order when delegated to different servers because the servers are not synchronized with respect to each other. Even requests to the same server may be reordered as described below.

Servers handle requests by iterating through all of their request lines and performing those requests with the appropriate flag. However, when there are multiple request lines per server-client pair we cannot guarantee that the requests will be performed in the order they are sent. Consider the case shown in **??**. A client writes a request to all but one of its request lines before the server handles the entire batch. Afterward the client writes requests to its last request line and then begins writing new requests to its first request line. Since the server handles requests in the order of the request line array, newer requests are handled before the oldest request in the last position.

For an application with non-commutative properties, the programmer can take care to delegate the entire order critical section. It that is unfeasible, a single request line preserves the ordering of requests made by a client to a specific server. Experimental results are shown with both 1 and 16 request lines per client-server pair to show the difference in throughput while maintaining ordering.

CHAPTER 3

EXPERIMENTAL EVALUATION

The following results shown are from a 28 core, 56 thread Intel Skylake machine with 97 GB of RAM. The system has three levels of cache, 32KB of L1 cache and 1,024KB of L2 non-inclusive and shared between hyperthreads on the same core. Our machine has 19,712KB L3 cache per socked shared among the 14 physical cores.

For spin, mutex, atomic, and flat delegation we use the number of threads available on the machine (56) unless otherwise stated. For trials with dedicated delegation *async* and *gepard* we list the number of servers. The balance of remaining threads are clients.

Like *Gepard*, asynchronous delegation threads are assigned to cores in a round-robin fashion. In the dedicated case all server threads are launched before any client threads. Memory for delegated data structures is allocated on the NUMA node corresponding to the server which it is assigned.

3.1 Fetch and Add

The experiment shown in Figure 13 allocates a user specified number of 64B variables shown on the x-axis. OS threads are launched, and for three seconds each thread selects a variable at random and then increments that variable by its synchronization technique. After three seconds the threads are joined and the throughput in Million Operations per Second (MOPS) is reported.

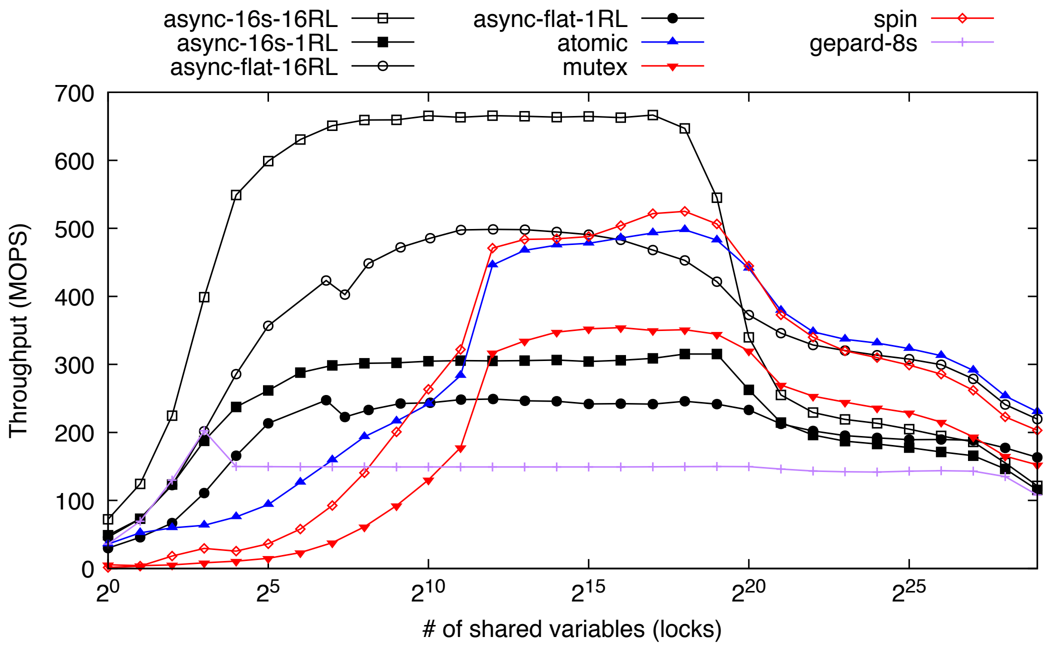

Figure 13. Throughput in MOPS by number of 64Byte variables. Higher is better.

The experiment was run with locking approaches, *mutex* and *spin*; atomic fetch and add (*atomic*); and delegation approaches (*Gepard*, *async-16s*, and *async-flat*).

The delegation approaches allocate memory as a two dimensional array, where the rows are the array dedicated to the individual server thread. **numa_alloc_onnode** is used to allocate the delegated data structures into dram local to the intended server's core. For flat delegation the number of shared variables is rounded up to the next highest multiple of the number of servers. For example 128 variables is rounded up to 168. For dedicated approaches when the number of shared variables is smaller than the requested number of servers the number of servers is reduced. For example a 16 server dedicated system handling 4 shared variables will run with 4 servers and 52 clients.

The atomic and locking approaches allocate memory using **malloc** as a single array.

3.1.1 Performance Under Contention

Delegation approaches excel for smaller numbers of shared variables. Notice dedicated delegation achieves over 600 MOPS for shared variable counts up to the size of the server cache. Flat delegation achieves consistent performance, topping out just under 500 MOPS. The reason for this even performance at low levels of shared variables is a lack of contention.

A variable is contended if multiple threads are trying to access it simultaneously. The chance that at least one pair of threads in the system is contending for a variable is plotted in **??** and described by $C = 1 - \frac{\binom{V}{T}}{V^T}$, where C is the probability of contention, V is total number of shared variables, and T is the number of threads. Of course the probability that all threads are contending is less than or equal to the probability that just one pair of threads are contending. We can see that the atomic and locking approaches do not reach peak throughput until after the probability of contention becomes trivial, near 2^{15} shared variables.

The fine grained locks and the atomic fetch and add have up to 56 threads accessing a single memory location. Figure 13 shows the atomic fetch and add and the fine grained locks reach peak performance as the probability of contention reaches 0 **??**.

In contrast, the delegation approaches, by design, have no contention for variables. When the number of shared variables is sufficiently small, the entire data set may be kept in a core's L1 and L2 cache. For the 16 server dedicated delegation case shown, each delegation server shares, as a hyperthread, a core with a delegation client. The combined overhead of the two threads is about 93 KB. This amount consumes the entire L1 cache and 62 KB of the L2. This leaves 962 KB of L2 cache for data. 960 KB translates to 15,000 variables per server, or 246,000 variables for the system. L2 cache should be exhausted at 2^{18} variables, which is where we see the performance begin to degrade.

Further, the portion of L3 cache available to the server is roughly $1/N_{serversonprocessor}$ of the L3 on the processor, or 2,464 KB. This translates to 39,000 additional variables per server, or 630,000 more variables for the system. L3 is exhausted when the number of shared variables approaches 2^{20}. Performance continues to degrade as the number of variables resident in DRAM increases and the likelihood of cache hits decreases.

3.1.2 Memory Subsystem

A common theme across all approaches is the degradation of performance beginning around 2^{20} shared variables. Due to the randomness in the benchmark, the probability that an approach will be reading from DRAM nears 1 as the number of shared variables increases. The probability, P that a variable picked at random in a system with cache size S_{cache}, variable size $S_{variable}$, total number of variables $N_{variables}$ is $P = \frac{S_{cache}/S_{variable}}{N_{variables}}$. As the probability that a variable is in cache approaches zero the time to access DRAM dominates the performance of the delegation server.

Much slower access to DRAM dominates the performance of all approaches in this region, however the degradation in performance is particularly extreme for dedicated delegation because accesses are made by a smaller subset of cores. Figure 14 shows as the number of server cores

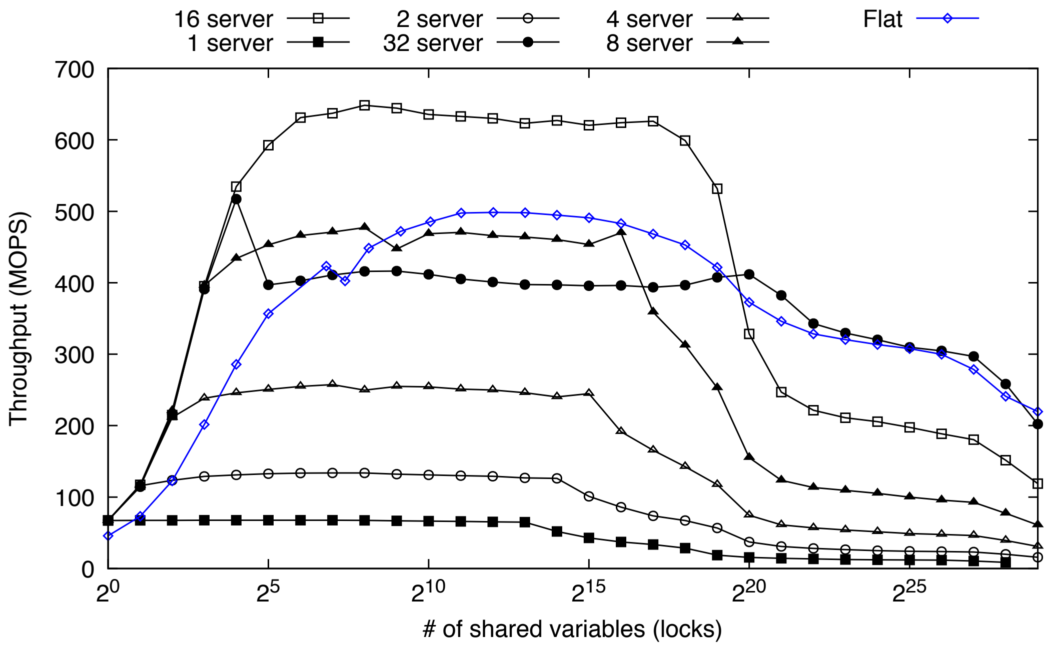

Figure 14. Throughput by Shared Variables by Number of Dedicated Servers

increases the tail performance of dedicated delegation improves. In Figure 15 we measure the number of loads from RAM for flat delegation vs a 16 server dedicated delegation and observe that the total number of ram loads for dedicated remains below those for flat though the RAM loads per operation scales throughout the range.

In a similar vein, a strength of flat delegation is its ability to utilize every core in the machine for DRAM access. Flat delegation also benefits from the lower cycle time for an on NUMA Node DRAM access. In comparison, the locking approaches will have a $1 : N_textsubscriptNUMANodes$ probability of accessing a variable at random that is addressed on the same node as the core.

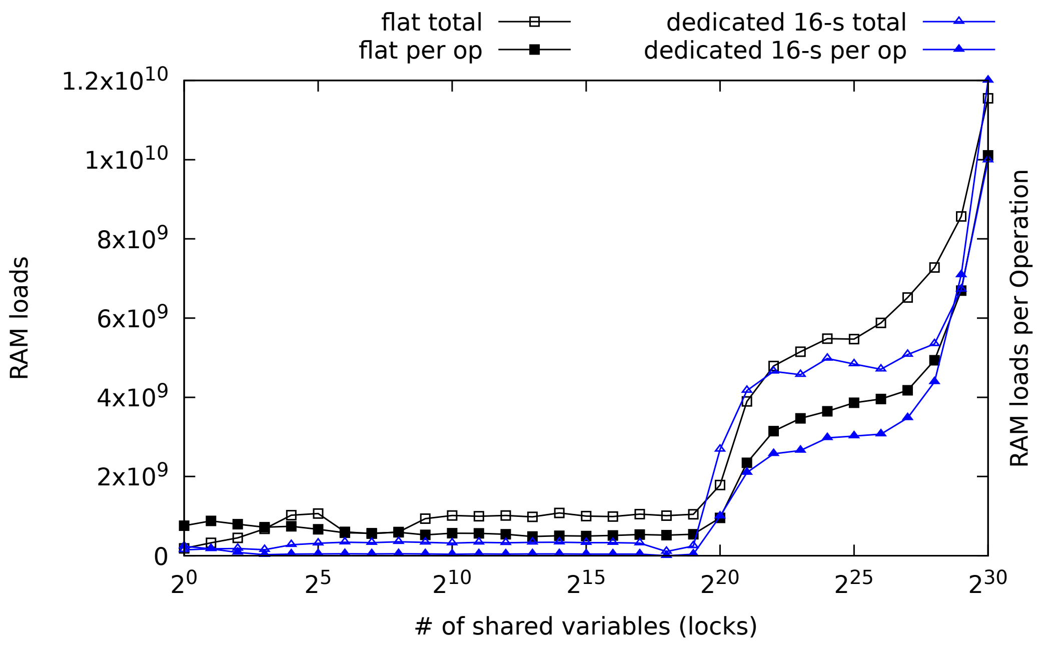

Figure 15. Delegation loads from RAM

CHAPTER 4

CONCLUSIONS

We have shown that asynchronous delegation approaches outperform the synchronous delegation approach of *Gepard* on the fetch and add benchmark. Further, asynchronous delegation can outperform fine-grained locking or atomic approaches when the delegated data structure fits within the server core's cache. Interestingly, flat delegation shows comparable performance to atomic and locking approaches as the size of the delegated data structure exceeds the size of the available cache.

CITED LITERATURE

VITA

NAME: NAME LASTNAME

EDUCATION: Ph.D., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2018.

M.Eng., Computer Engineering, University of Illinois at Chicago, Chicago, Illinois, 20xx.

B.Eng., Computer Engineering, University of Illinois at Chicago, Chicago, Illinois, 20xx.

ACADEMIC EXPERIENCE: Research Assistant, Computational Population Biology Lab, Department of Computer Science, University of Illinois at Chicago, xxxx - 2018.

Teaching Assistant, Department of Computer Science, University of Illinois at Chicago:

- Computer Algorithm I, Spring xxxx and Fall xxxx.
- Secure Computer Systems, Fall xxxx