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

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

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. Arrays for message passing are augmented from 4 to 16 indices per server-client pair providing for even greater concurrency. 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 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

Parallel programs traditionally follow a model where multiple threads perform operations on the same data structure. This model creates the opportunity for a simultaneous access of shared memory by multiple threads that yields a non-deterministic result, or data race. The commonly implemented solution is to synchronize access to shared data structures through mutual exclusion (mutex) locks.

Mutex locks are implemented using locked atomic instructions. 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 [**?**].

A mutex lock works by 'claiming' a variable using an atomic instruction like LOCK CMPXCHG. If a thread is successful in writing to the variable it can proceed to operate on the critical section of shared memory, and then release the lock. If the thread is unsuccessful it will wait by some policy until it successfully claims the lock variable. This programming pattern serializes accesses to the shared memory therefore eliminating 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 by Roghanchi et. al. in *FFWD* [**?**], grants exclusive control of a data structure to a single thread called a *server*. *Client* threads delegate operations on data structures to the appropriate *server* by passing a message called a request. The *server* is one thread, performing one operation at a time. The delegated operations are serialized and the data race eliminated.

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 requested parameter values into the appropriate registers and

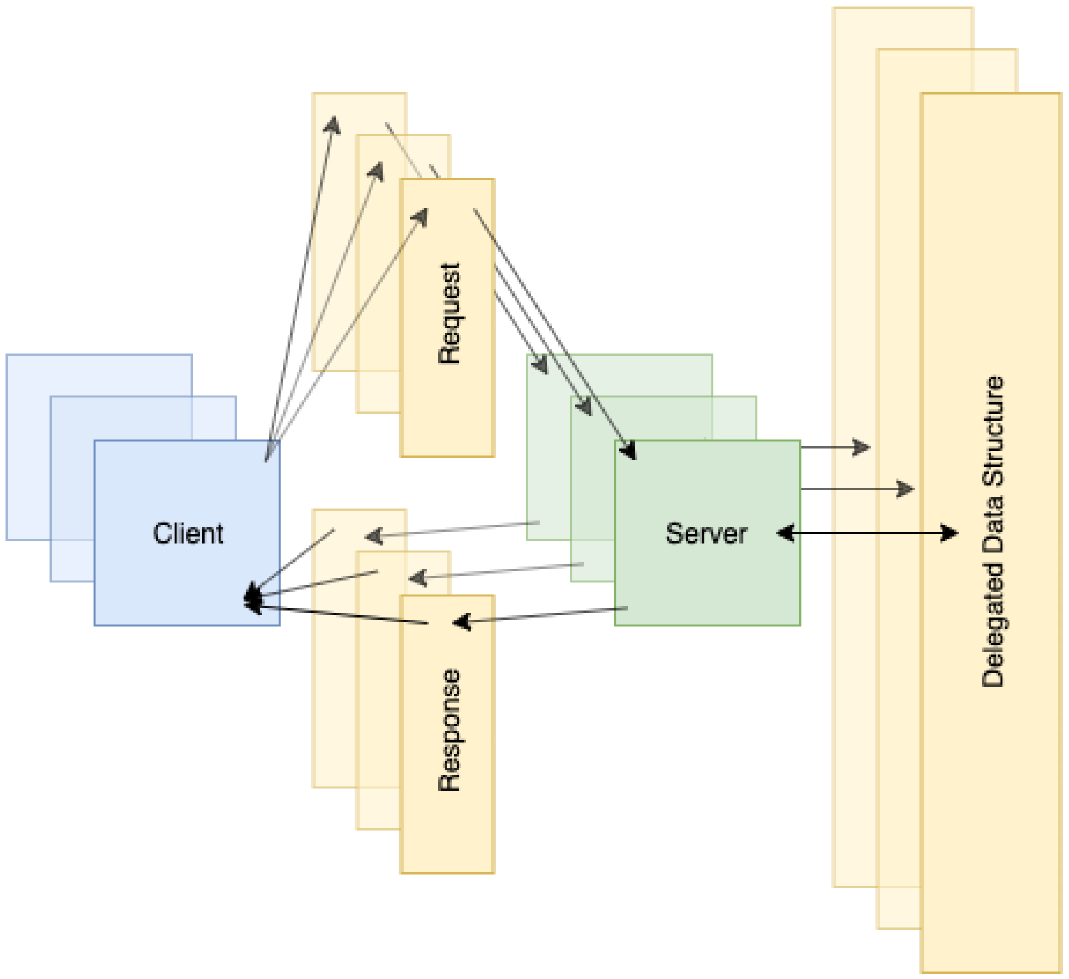

Figure 1. A delegation system with 3 clients and 3 servers. The foreground client makes a request to all three servers.

then calls the function pointer. The server stores the return value from the function and the flag variable into the corresponding index in the array of responses.

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. A small delegated data structure may fit within a server's on-core cache and remain resident for the duration of the program. In contrast, a system with multiple physical cores accessing the same data structure will share cache lines, greatly reducing the likelihood of a high 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 interface by enabling a thread to switch to productive work through the use of fibers. Based upon *libfiber* [?], *Gepard* fibers are light-weight, cooperatively scheduled user mode threads. *Gepard* threads run a fiber manager overseeing multiple client fibers. A client fiber writes a delegation request to the server then yields, thereby invoking the fiber manager which switches context to another client fiber. After some time the original client fiber will be reactivated and continue execution. The major advantage of *Gepard* is that it enables a single thread to engage with multiple servers concurrently with the effect of increasing throughput despite constant individual request latency.

CHAPTER 2

ASYNCHRONOUS DELEGATION

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.
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

By decoupling a delegation call from its response, asynchronous delegation is able to introduce more concurrency than *Gepard* and achieve greater system throughput. Operations which *Gepard* performs after a call to **delegate(s, retval, f, args)** are now done in a callback function invoked upon receipt of the response. The callback, if required, is passed to the delegate function in place of the return value as shown by **Delegate_Async(s, cb, f, args)**. An excerpt of the asynchronous delegation API is shown in 2.

The change in API allows asynchronous delegation to completely remove *libfiber*, the agent of client side concurrency in *Gepard*, along with *libfiber's* associated memory and computation overhead. *Gepard* fiber managers keep track of a fiber's state which includes its call stack and instruction pointer. In contrast, asynchronous delegation tracks a client's request leaving the restoration of state to the callback function. By a combination of memory footprint and compuation overhead in contexting switching, *Gepard's* throughput peaks at 64 fibers / thread. On the other hand, we show asynchronous delegation efficiently tracks well over 1,000 concurrent, outstanding pending requests in our benchmarks.

The reserve of pending requests is available because calls to **Delegate_Async()** are non-blocking for most invocations. The client function can continue to generate requests while interaction with the delegation server is deferred until the most optimal time depending on the delegation strategy.

In this chapter we explore the opportunities that this explosion of available requests provides to increase concurrency in asynchronous delegation with one or more dedicated server threads.

Later, we show how asynchronous operation can be applied to flat delegation, a more recent, serverless design.

Figures in this chapter are generated with variations of a fetch and add benchmark. In this benchmark (1ea) 64B, variable is allocated for each delegation server in the system. For a period of three seconds, clients select a server at random and delegate an increment function to the server. The function increments the server's variable and returns. There is no callback. Results shown are the harmonic mean of ten samples. Unless otherwise stated, asynchronous delegation is configured with 16 request lines per client - server pair and a 32 length pending request queue for each server.

2.1 Asynchronous Dedicated Delegation

Figure 2 sketches an asynchronous dedicated delegation system with 1 client and 3 servers. The client generates requests and places them in the fixed-length request queue for the required server. When any of the queues reaches capacity, the client suspends request generation and begins the process of writing out requests to servers. The client polls its response lines one by one, executing the callback function if provided. When a response is present a request is popped from the request queue and placed in the corresponding request line. After all response-request lines have been handled **Delegate_Async()** returns.

To use asynchronous dedicated delegation as shown in Figure 2, the user first initializes 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**

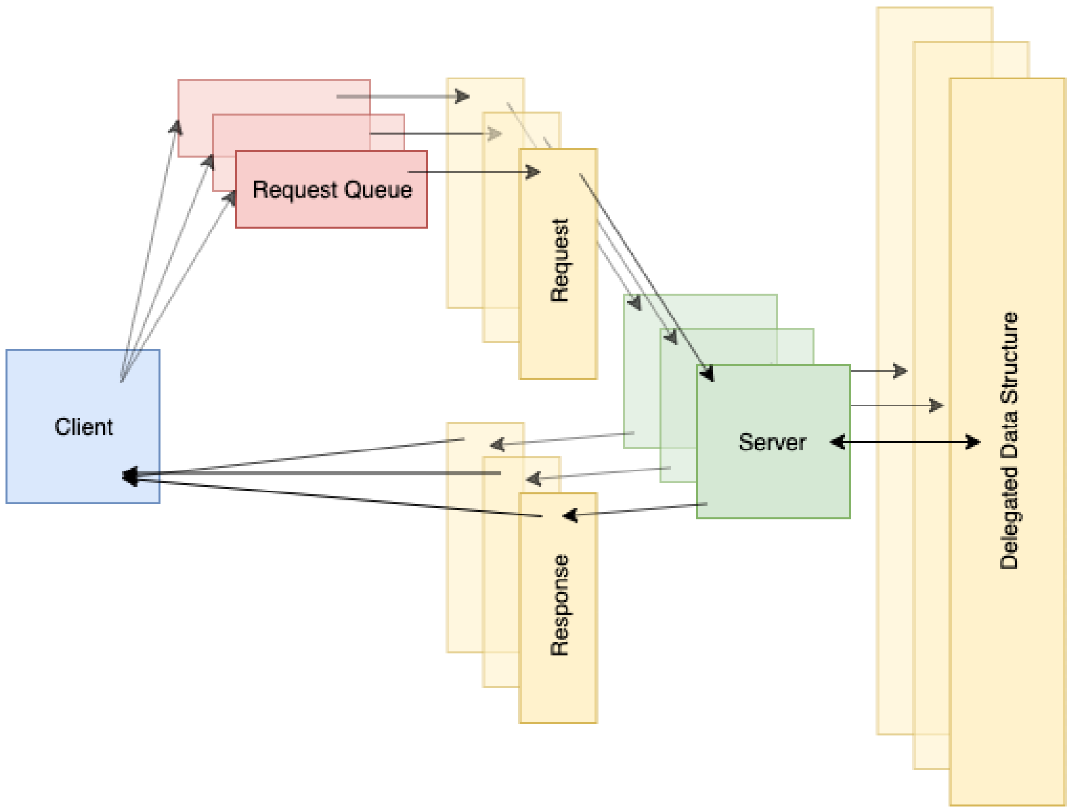

Figure 2. An asynchronous delegation system with 1 client and 3 servers in our highest throughput configuration. The foreground client makes a request to all three servers. Requests are written to a pending request queue which is periodically flushed out to the request line.

first initializes an empty queue of pending requests for each running server on the NUMA node which the client thread is assigned. 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, a push to the pending request will fail when the queue is full. At this time the client iterates through its entire array of responses. If a response is ready, the client invokes the callback with the corresponding return value, and then writes a new request, popped from the appropriate pending request queue, to the corresponding request line.

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

2.1.1 Hurry Up and Wait - The Pending Request Queue

A counter-intuitive feature of the asynchronous API is the ability to hold a large number of pending requests in reserve until the optimal time to write them to the server. One might expect writing requests as quickly as possible to a constantly cycling server would yield the maximum throughput. However, Figure 3 provides evidence the reserve has a profound impact on overall system throughput.

The experiment shown in Figure 3 plots the throughput of our benchmark with a reserve queue and an asynchronous dedicated delegation configuration with direct writes to the server's request line. The direct write program tracks the index of the last request written to each server, writing its current request to the next available index. If the next index is not available the program polls the response line until it is available. Although direct write achieves comparable

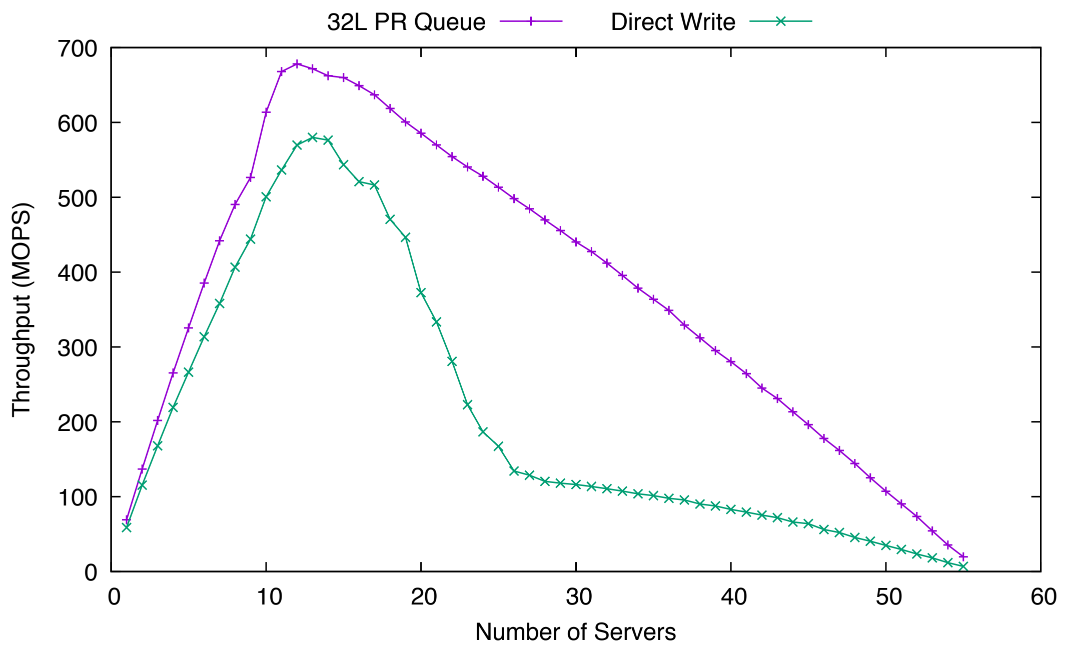

Figure 3. Throughput in MOPS for a 56 thread system by number of servers with a 32 length pending request queue and direct request line writes.

server consumption rate to the program with pending request reserve, it shows undesirable degradation in performance when the number of clients bounds the throughput of the system.

We implement 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 and writing requests from the pending request queue to the server if available.

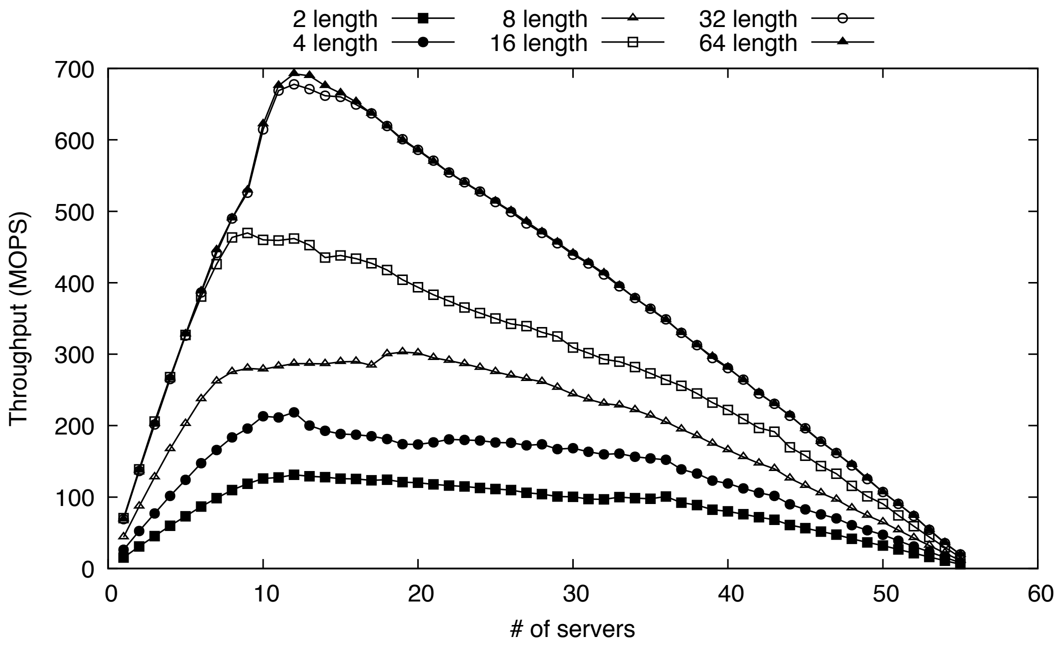

Figure 4. Throughput in MOPS for a 56 thread system by number of servers by pending request queue length. The remainder of threads are clients.

How long should the individual pending request queues be? From Figure 4 we see that the length of the pending request queue has a clear impact on overall throughput. Figure 4 shows our familiar benchmark plotted with various pending request queue lengths from length 2 to length 64 on a asynchronous dedicated delegation configured with 16 request lines. 2 - 8 length queues exhibit poor performance because queues smaller than the number of request lines are unable to utilize the entire capacity of the server.

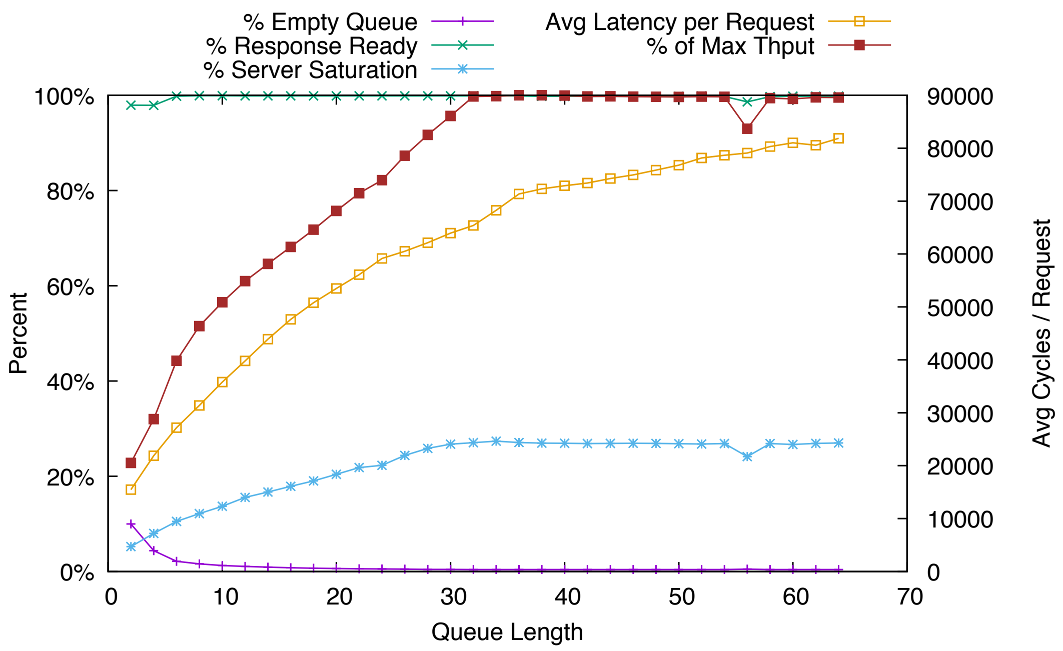

Figure 5. 16 Request Line Configuration: Latency, Server Saturation, Response Readiness, and Failed Queue Pops by queue length.

Interestingly, Figure 4 shows that queue lengths greater than or equal to 16 show the same client consumption rate when the system is bound by server capacity; suggesting higher server saturation increases server throughput. This relationship is confirmed by Figure 5. Figure 5 plots throughput, server saturation, response ready rate, proportion of failed pending request queue pop operations, and average request latency with our microbenchmark using 16 servers and 40 clients. We see that peak system throughput is correlated with peak server saturation.

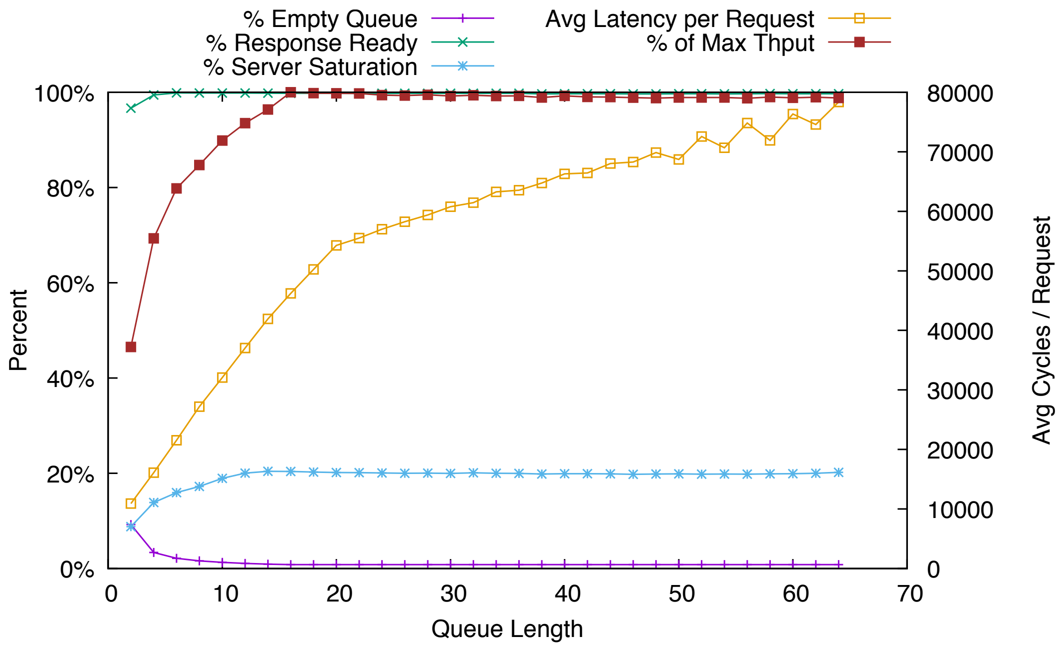

Figure 6. 8 Request Line Configuration: Latency, Server Saturation, Response Readiness, and Failed Queue Pops by queue length.

From Figure 4 we also observe the marginal return on throughput for longer queue lengths diminishes after length 32, or $2 * N_{requestlines}$. For more insight into this, we again reference Figure 5. Server saturation increases and failed queue pop instances decrease with the extension of the pending request queue. At roughly $2 * N_{requestlines}$ the gains cease. This finding suggests that the probability of having enough requests in all queues to fill all request lines approaches 1 somewhere near $2 * N_{requestlines}$. Figure 6 and Figure 7 plot the same experiment configured with 8 and 4 request line servers respectively. Both support the $2 * N_{requestlines}$ hypothesis.

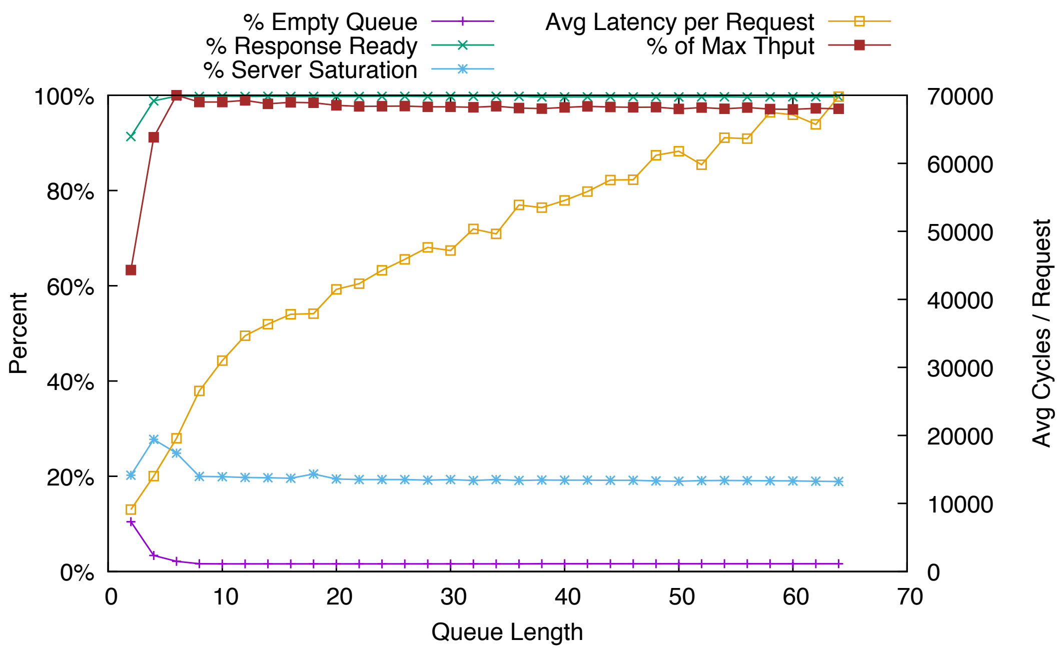

Figure 7. 4 Request Line Configuration: Latency, Server Saturation, Response Readiness, and Failed Queue Pops by queue length.

Based on these results, 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.

2.1.2 More Concurrency with Request Lines

Correlated to the length of the queue is the number of request and response lines used to pass messages between clients and servers. *Gepard* as tested in chapter 3 is configured with 4 request lines per client. *Gepard*, however, manages far fewer concurrent requests due

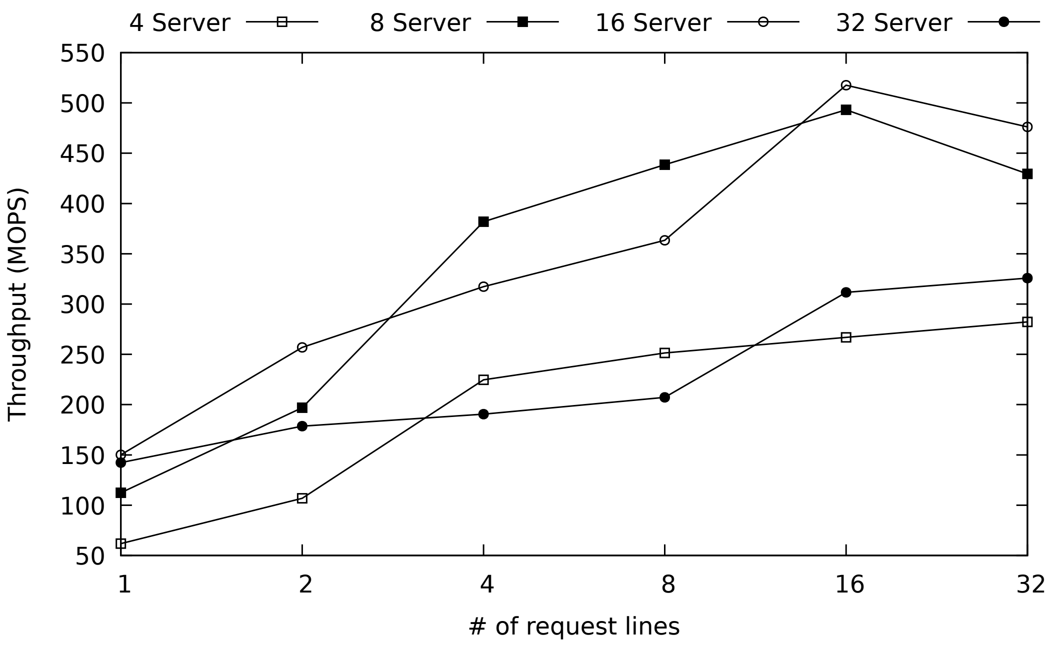

Figure 8. Throughput in MOPS for a 56 thread system by number of request lines.

to the overhead of the fiber library. The increased rate of request production in asynchronous dedicated delegation provides the opportunity to experiment with writing more concurrent to requests to the servers.

Figure 8 shows the measured system throughput for our benchmark with throughput plotted against the number of request lines per server-client pair. Figure 8 shows that increasing the number of request line generally increases the throughput until about 16 request lines when performance degrades.

Do I know why?

Due to this observation we use 16 request lines for the rest of our experiments.

2.1.3 Client and Server Production Rates

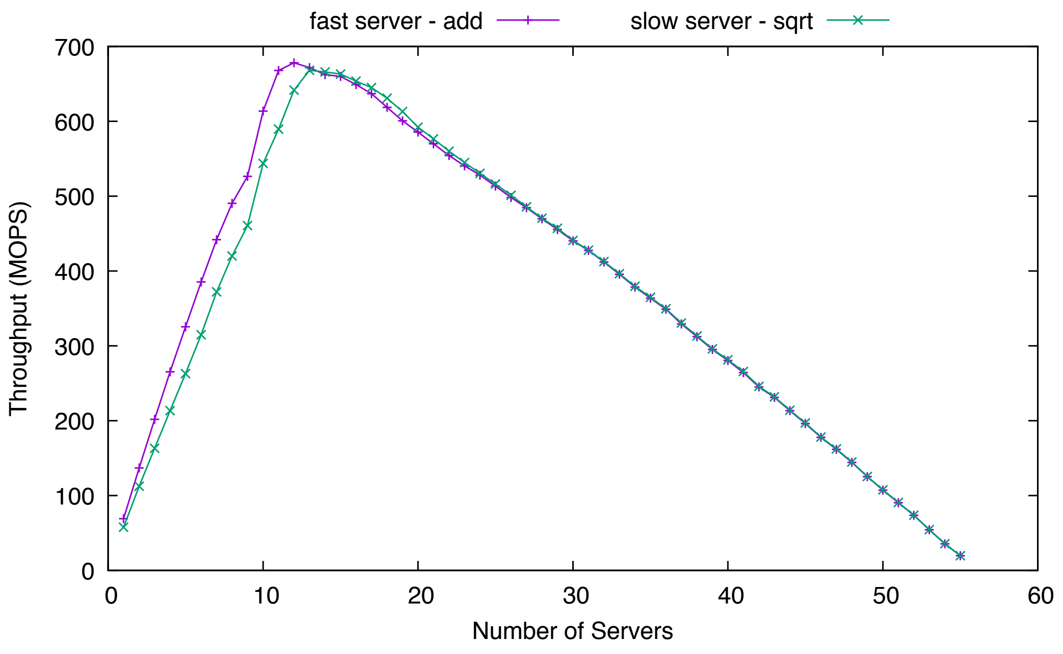

Figure 9. Throughput in MOPS for fetch and add vs fetch and sqrt. While server speed is faster than client speed, peak throughput will require more servers when the delegated function is slower.

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.

Equation 2.1 describes this rule of thumb, 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 $N_{client} * Thput_{client}$.

$$Thput_{system} = \min(N_{server} * Thput_{server}, N_{client} * Thput_{client}) \tag{2.1}$$

Figure 9 displays results of an experiment where we compare the throughput by number of servers of a faster (addition) and slower (sqrt) delegated function for our benchmark. In the regions left of peak throughput shown in Figure 9 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 the equilibrium point the system is bound by client production. The downward slope corresponds to the $\mathrm{Thput}_{\mathrm{client}}$.

Figure 9 shows Equation 2.1 to be a good model. Consumption rate slows for the server performing 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 for the slow function than it does for the faster one, suggesting that programmers should increase the number of servers when the delegated function is more computationally intense.

2.2 Asynchronous Flat Delegation

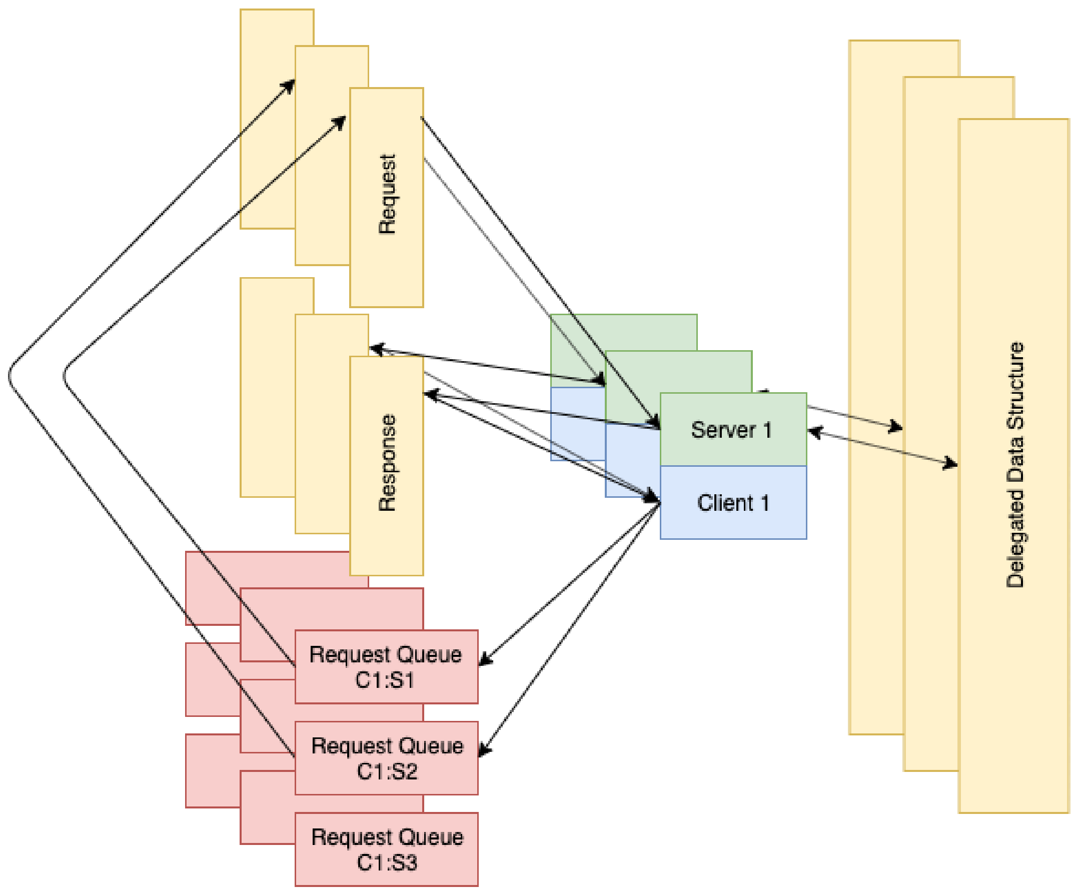

Figure 10. A flat delegation system with 3 client-server threads. The foreground client makes a request to itself and the background client.

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 10 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 10 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 writing out requests. Flat delegation servers run periodically as determined by the scheduling policy instead of continuously as they do in the dedicated approach. We continued to experiment with writing requests directly to the servers as described §2.1.1 because of the periodic server invocation.

2.2.1 Scheduling Strategies for Asynchronous Flat Delegation

Figure 11 summarizes the throughput for the asynchronous flat delegation scheduling strategies tested on our benchmark. The independent variable, however represents the total number of threads running, e.g., at 20 there are 20 total threads, each running as both client and server. The parameters for each are described in the following paragraphs. The server function

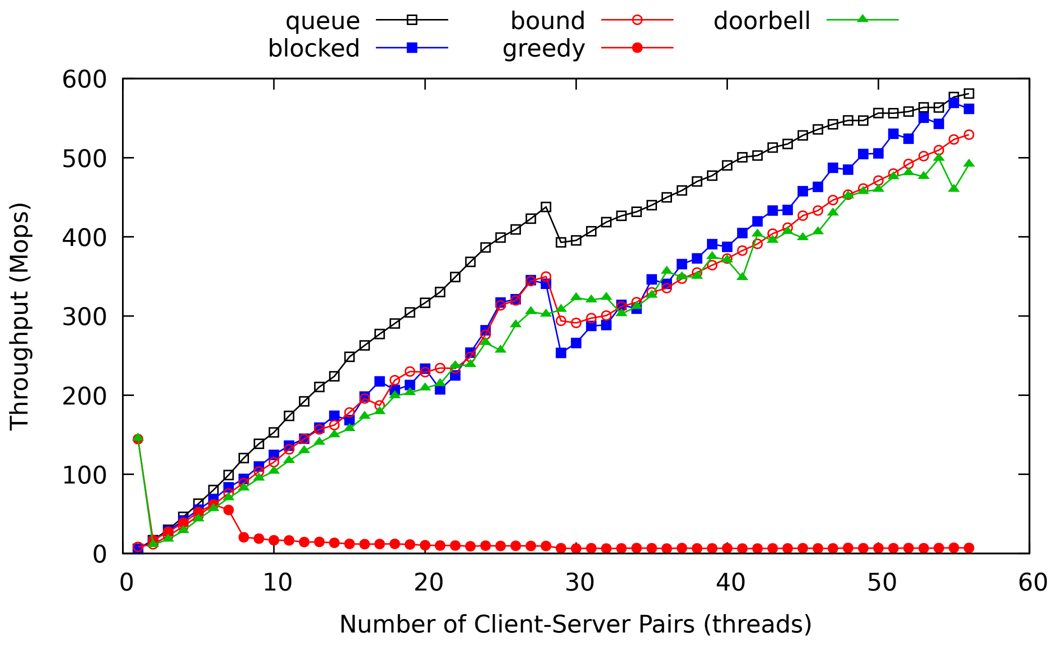

Figure 11. Flat delegation throughput by scheduling policy.

is the same server function as Asynchronous Dedicated Delegation and *Gepard* except it iterates through its request lines once and then returns.

Holding requests in reserve 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 12 and Figure 13 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. We should note that average request latency is on the

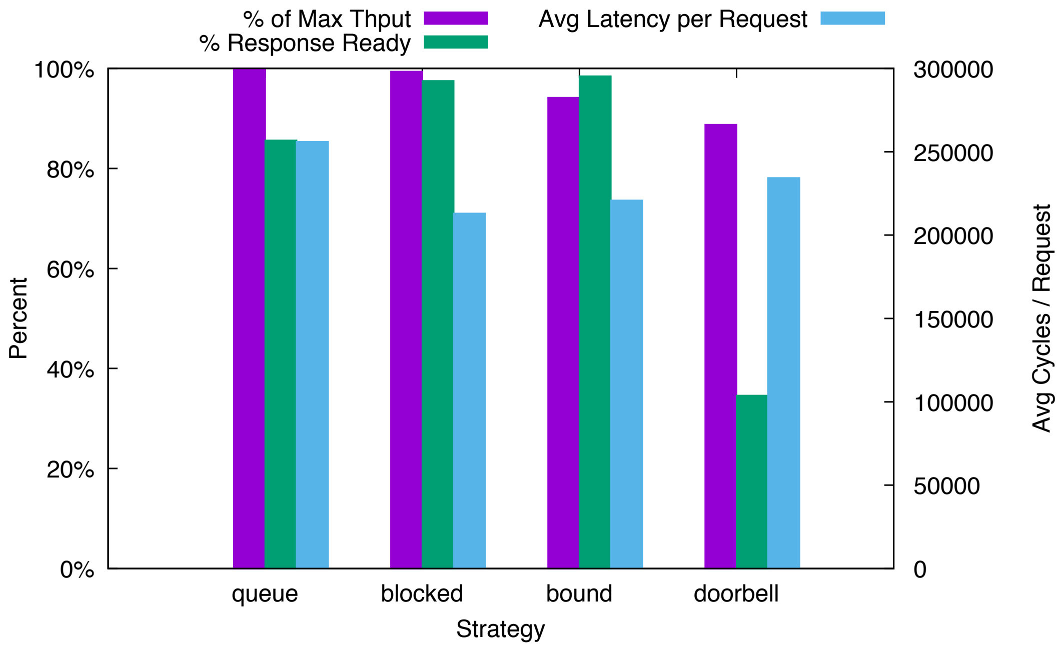

Figure 12. 56 server-client pair comparison of average request latency, response line availability, and comparative throughput. Greedy omitted for scale.

order of 10 times greater for asynchronous flat delegation than it is for asynchronous dedicated delegation shown in Figure 5 .

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 11 pending request queue is the same as direct delegation described in subsection 2.1.1 but with the server function invoked prior to writing and reading

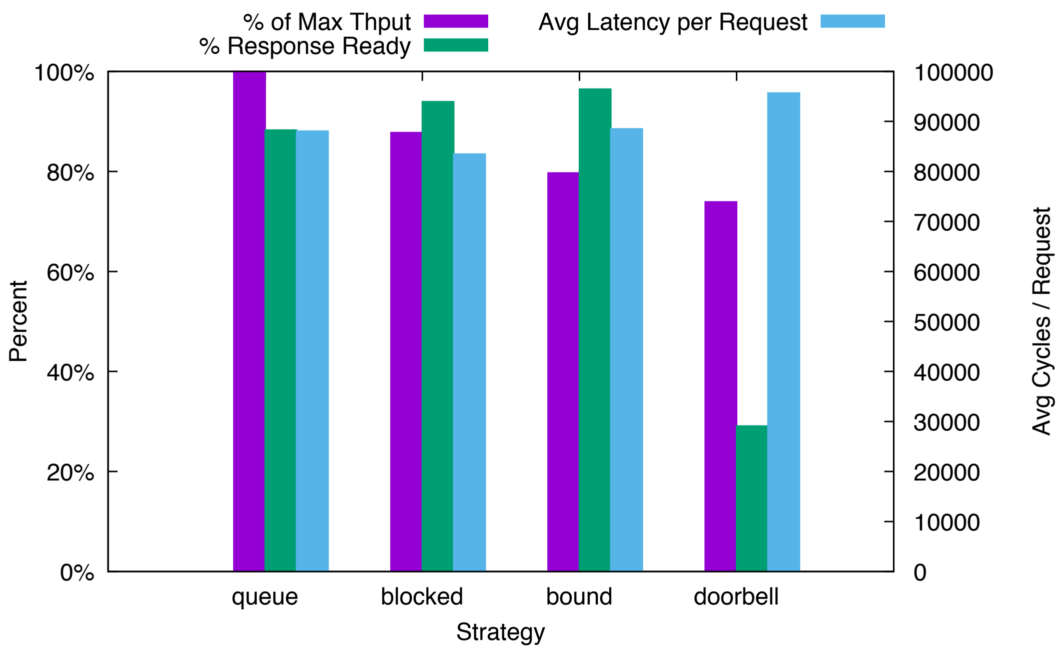

Figure 13. 28 server-client pair comparison of average request latency, response line availability, and comparative throughput. Greedy omitted for scale.

request lines. This approach outperformed all others in our trial, and is the approach tested in chapter 3.

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 Figure 11 *blocked* shows comparable results to *queue* for larger numbers of threads. *Blocked* cuts down on the delegation memory footprint because it eliminates the pending request queues. For larger systems, or systems with smaller amounts of on core cache *blocked* may be preferable for better on-core cache utilization.

The *blocked* strategy exhibits a desirable property shown in Figure 12 and Figure 13. The latency for any individual request, from generation to execution of the callback function is up to 15.5% 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. The doorbell approach measures the highest average request latency, likely due to the latency of writing the doorbell message to a remote thread.

2.3 Ordering Guarantees

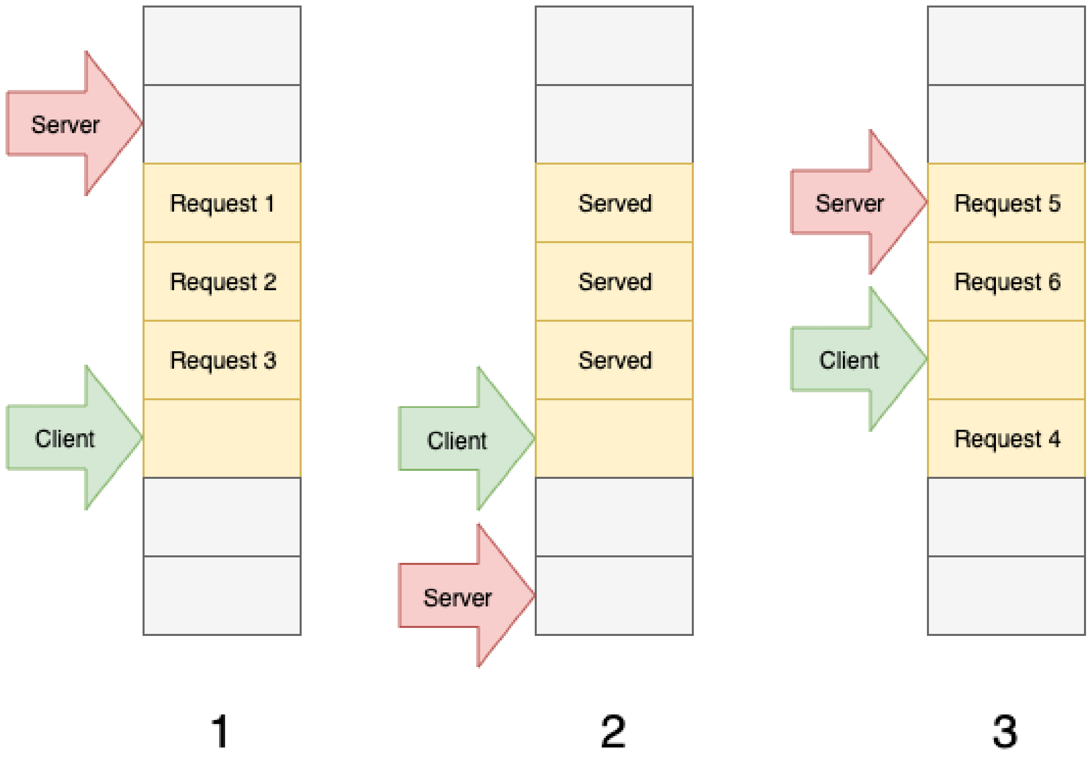

Figure 14. The server may execute requests out of order when multiple requests lines are used by one client. (1.) The client issues 3 requests before the server reaches its section of the request line array. (2.) The server handles all requests in this clients section before the client writes a fourth 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 Figure 14. 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, albeit with detrimental impacts to overall throughput. Experimental results are shown with both 1 and 16 request lines per client-server pair to show the difference in throughput while maintaining ordering between delegated requests to the same server.

CHAPTER 3

EXPERIMENTAL EVALUATION

The results shown are from a 28 core, 56 thread Intel Skylake machine with 97 GB of RAM. The system has three levels of cache: On-core non-inclusive L1 and L2 of size 32KB and 1,024KB respectively, and 19,712KB of L3 cache per socket shared among the 14 physical cores on one processor.

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 processors in a round-robin fashion. For example, on our machine threads 0-13 are on processor 0 and 14-23 are on processor 1. Threads are assigned in the order 0, 14, 1, 15, etc... 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 processor which it is assigned.

3.1 Fetch and Add

The experiment shown in Figure 15 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

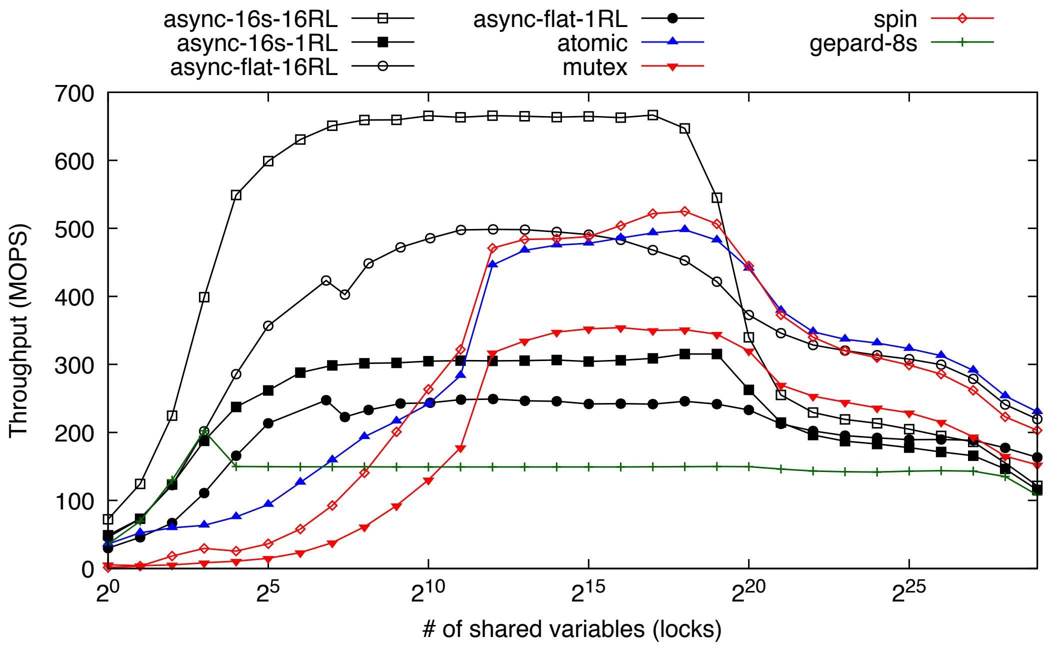

Figure 15. Throughput in MOPS by number of 64Byte variables. Higher is better.

seconds the threads are joined and the throughput in Million Operations per Second (MOPS) is reported.

The experiment is run using the POSIX threads implementations of *mutex* and *spin* locks, the gcc compiler intrinsic atomic fetch and add (*atomic*), and delegation approaches (*Gepard-8s*, *async-16s*, and *async-flat*).

The delegation approaches allocate memory as a two dimensional array, where the rows of the array are assigned 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's processor. E.g., server 1 is assigned to core 14 on processor 1, NUMA node 1.

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. By the pigeonhole principle, we can be guaranteed that when the number of shared variables is less than the number of threads, the number of threads contending for variables is at least $2*(T-V)$ where V is total number of shared variables and T is the number of threads. The impact of the contention is evident for the locking and atomic approaches, which demonstrate consistently low throughput until the number of shared variables increases beyond the number of threads. The probability of contention wanes as the number of variables grows. The chance that at least one pair of threads in the system is contending for a variable is described by $C = 1 - \frac{\binom{V}{T}}{V^T}$ where

C is the probability of contention. C drops below 10% at 2^{14} and is practically zero at 2^{19}, which corresponds with the peak throughput of of the locking and atomic approaches.

An advantage of delegation approaches, previously shown by *Gepard*, is the lack of contention for variables. Figure 15 shows that asynchronous approaches meet or exceed the throughput gains made by *Gepard* for small numbers of variables.

Interestingly, asynchronous dedicated delegation outperforms all approaches on the benchmark up to 2^{20} shared variables. This is no accident, when the delegated data structure is sufficiently small, it remains resident in the server's on-core cache. For the 16 server dedicated delegation case shown, each delegation server shares, as a hyperthread, a core with a delegation client. The combined memory overhead of delegation for the two threads is about 93 KB. This amount consumes the entire L1 cache and 62 KB of the L2 leaving 962 KB of L2 cache for the delegated data structure. 960 KB translates to (15,000) 64B variables per server, or 246,000 variables for the system. L2 cache is exhausted at 2^{18} variables, which is where Figure 15 shows degradation of asynchronous dedicated delegation throughput performance.

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.

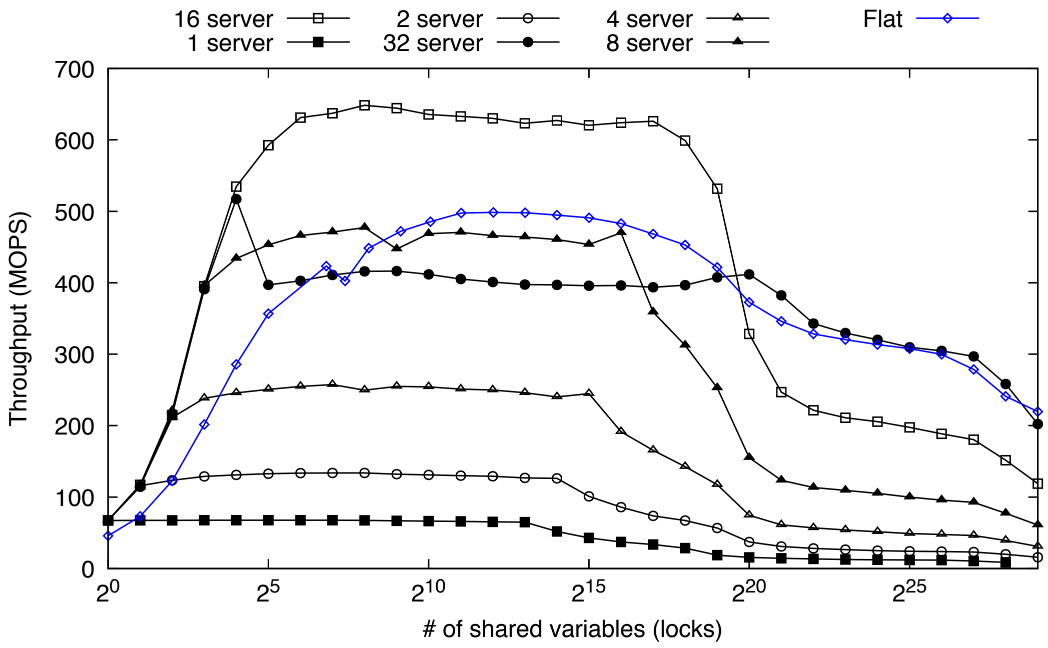

Figure 16. Throughput by Shared Variables by Number of Dedicated Servers

3.1.2 Memory Subsystem

All approaches display a reduction in throughput beginning around 2^{20} shared variables. This is because the likelihood a shared variable will be in cache decreases as the number of shared variables increases. Because of the randomness in the benchmark, we can estimate the probability that a shared variable will be DRAM resident. 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 $N_{variables}$ follows Equation 3.1

$$P = \frac{S_{cache}/S_{variable}}{N_{variables}} \tag{3.1}$$

Since S_{cache} is a fixed property of the hardware, we see the probability that a variable is in cache approaches zero with an increasing number of shared variables.

Much slower access to DRAM dominates the performance of all approaches as the number of shared variables grows beyond 2^{20}. For dedicated delegation the degradation in performance is particularly extreme because accesses are made by a smaller subset of cores than the other approaches. This observation is substantiated by Figure 16, which shows that tail throughput rises with the number of delegation servers. Cross referencing Figure 15, we see that the 32 server asynchronous dedicated delegation and flat delegation shown in Figure 16 achieve nearly the same throughput as the locking and atomic approaches on our benchmark.

Our final observation of flat and high server count delegation systems provides motivation to continue optimizing the systems for use in large delegated data structures. We delegate data structures onto the same NUMA node in which the server thread is running. As a result delegation servers will access NUMA local dram for 100% of delegated operations. In comparison, a data structure with synchronized access through locks or atomic operations will have a $1/N_{nodes}$ chance of performing a local operation where N_{nodes} is the number of NUMA nodes in the system.

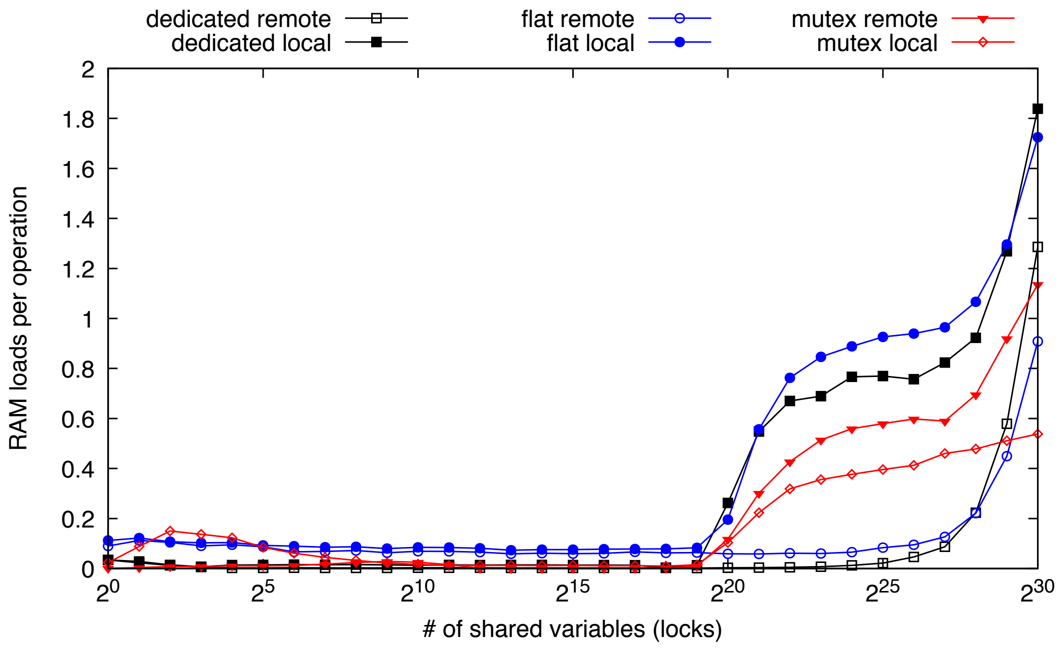

Figure 17. Delegation loads from RAM

We can observe the NUMA effect in Figure 17 where local and remote DRAM loads were measured using `perf stat` for dedicated delegation in blue, flat delegation in black, and mutex locks in red. Local DRAM loads have filled in pips while remote loads are outlined. We observe in Figure 17 a wide separation between remote and local DRAM loads per operation for delegation approaches. As expected, mutex shows a balance much closer to an even split between nodes.

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 is shown to outperform fine-grained locking and atomic approaches when the delegated data structure fits within the delegation server thread's on-core 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.

Opportunities for continuing work remain. The performance of asynchronous delegation should be measured on a variety of microbenchmarks to confirm these initial findings. Asynchronous delegation should also be ported into a real world application and its impact on performance measured.

Flat delegation shows promise to equal or outperform synchronization approaches throughout the range of shared variables. More research is necessary on the strategy for invoking the server function on a flat delegation thread with the goal of finding an equilibrium between client and server functions that maximizes overall throughput.

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