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

Defense Committee:
Prof. Jakob Eriksson, Chair and Advisor
Prof. Xingbo Wu
Prof. William Mansky

ACKNOWLEDGMENTS

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

YOUR INITIAL

PREFACE

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMS	American Mathematical Society
CPU	Central Processing Unit
CTAN	Comprehensive TEX Archive Network
FIFO	First In First Out
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 two designs for asynchronous delegation, dedicated and flat. 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.

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.

Further, we examine the effects of hardware on the performance of asynchronous delegation and provide guidance to programmers for tuning their systems.

CHAPTER 1

BACKGROUND AND MOTIVATION

1.1 Synchronization

Threads operating in shared memory space 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.

For access to individual words, locked atomic instructions provide a guarantee that a write will be seen consistently across all threads. The guarantee is made by locking the system bus or in 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 successful write to the lock variable using LOCK CMPXCHG a thread can proceed to perform a more complicated critical section using non-atomic instructions, and then release the lock. The effect is accesses to the shared memory are serialized and threads without access to the lock end up waiting until the lock is available to do useful work.

1.2 Delegation

Delegation, on the other hand, provides exclusive control over a block of memory addresses to a single thread called a *server*. Servers receive requests to perform an operation on their

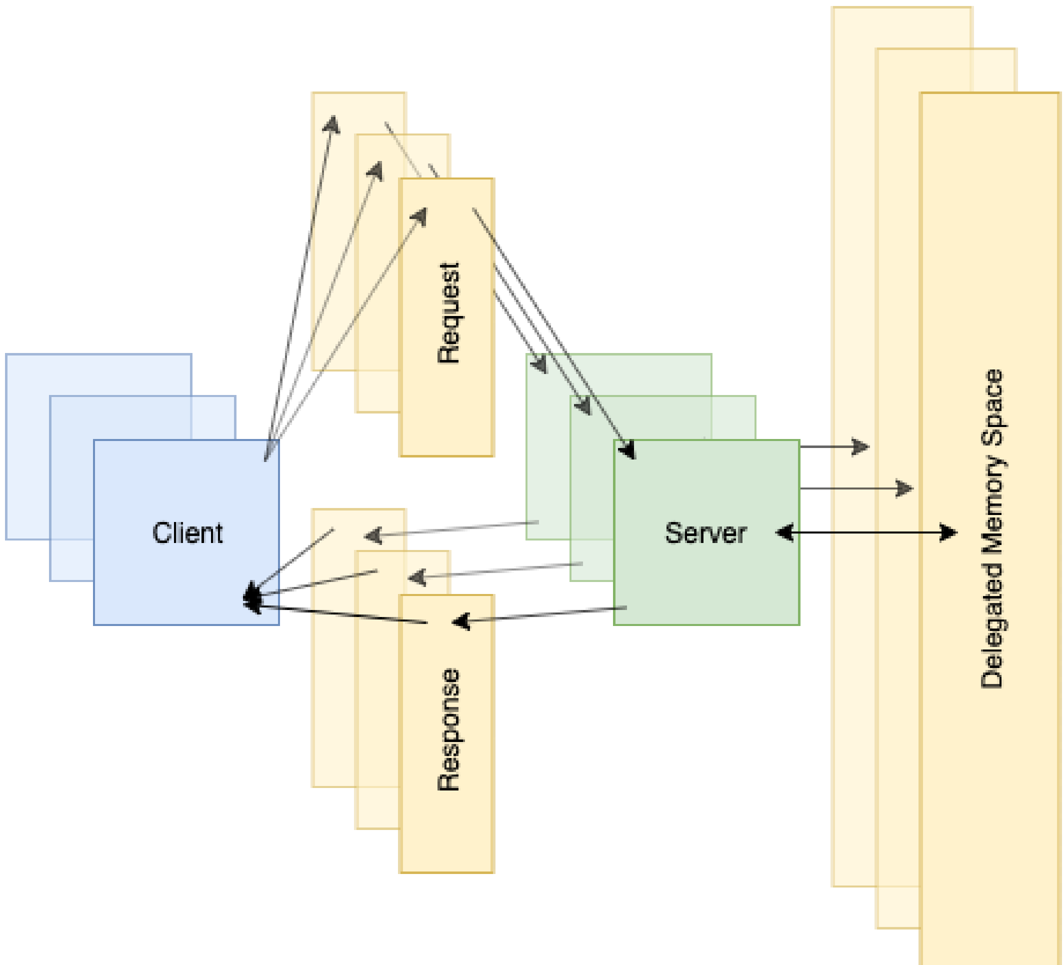

Figure 1. A delegation system with 3 clients and 3 servers. The foreground client makes a request to all three servers.

memory from other threads called *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 in its delegated memory space, 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.

Each delegation *server* iterates through an array of requests. When a new request is encountered the server moves the variables included in the request to the appropriate registers and then calls the function pointer. The server stores the return value from the function and the flag variable into an array of responses.

Clients maintain an array of request pointers targeting the client's assigned request line on each server. *Clients* also maintain an array of pointers to the servers' responses. A client identifies a response is ready when it reads that the flag in the response is equal to the value it was set to in the request.

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. 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 then begin the cycle over again. Although the latency of a single request remains nearly constant, the overall throughput is increased by issuing parallel requests.

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 DESIGN

2.0.1 General

The change offered by asynchronous delegation is evident in the change to the API. 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)** where cb is a callback function with the return value as a parameter. For most invocations a call to **Delegate_Async(s, cb, f, args)** is non-blocking. The entire API of asynchronous delegation is shown in Table I.

The strategy for issuing requests to the delegation servers varies in the asynchronous designs detailed below.

2.1 Dedicated Delegation

To use dedicated delegation 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 **Delegate_Thread_Create**. When a thread is started with **Delegate_Thread_Create** an empty queue of pending requests for each running server is allocated on the client's NUMA node.

Launch_Servers() Starts a server thread, allocates and initializes the request and response lines.

Delegate_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.

Within the application code the programmer uses **Delegate_Async** to enqueue a pending request. 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 upon return of the application code.

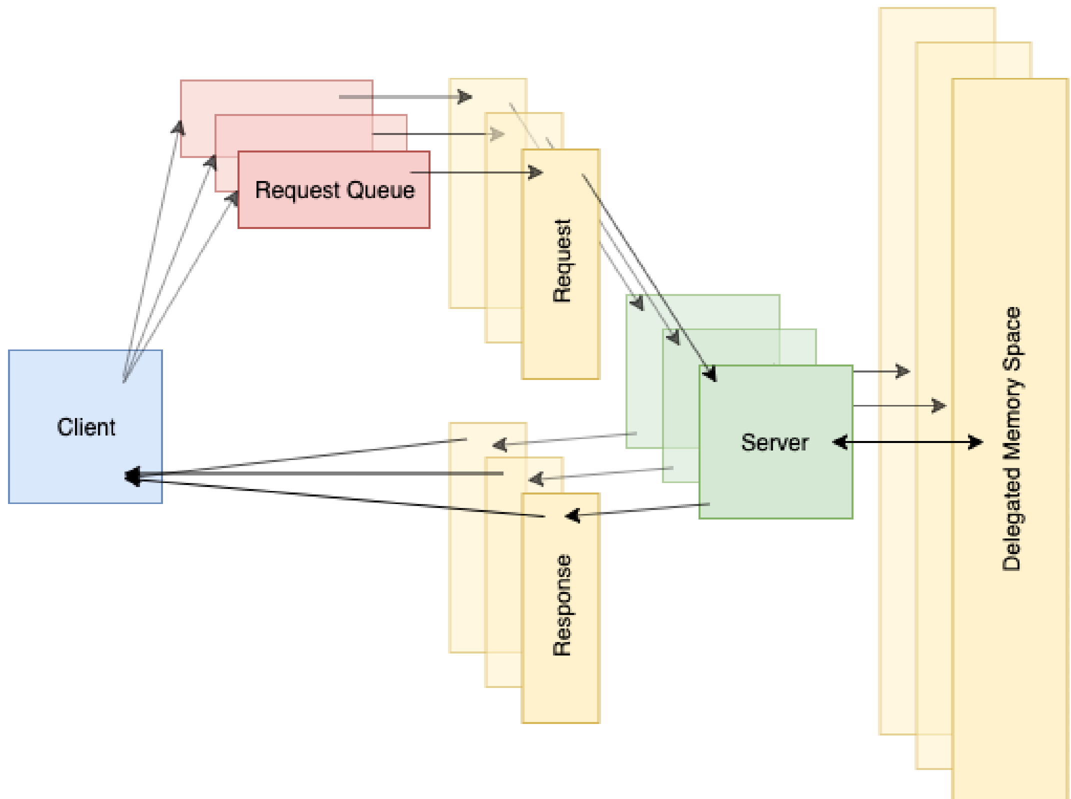

Figure 2. A delegation system with 1 client and 3 servers. The foreground client makes a request to all three servers. Requests are written to a pending request buffer which is periodically flushed out to the request line.

2.2 Flat Delegation

Flat delegation runs clients and servers on the same thread. There is no call to **Launch_Servers**, instead servers are initialized upon the call to **Delegate_Thread_Create**. Figure 3 illustrates

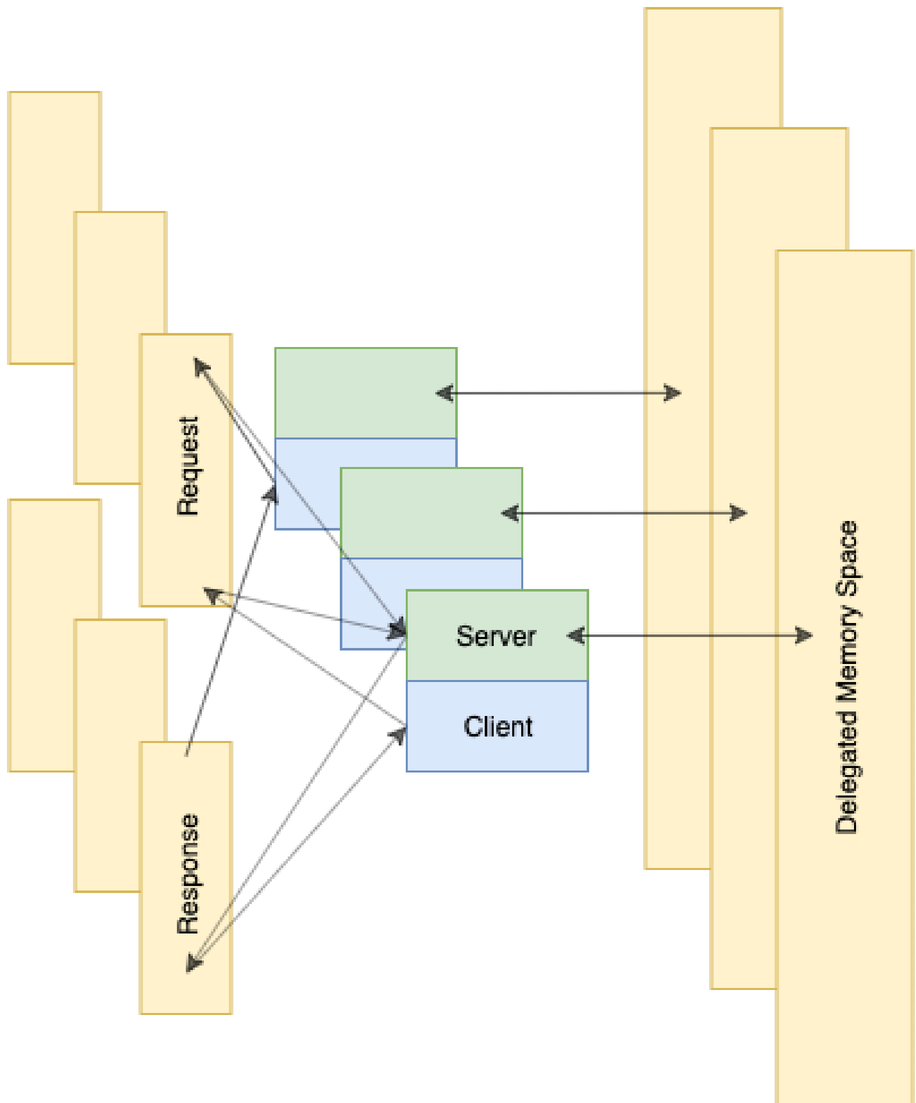

Figure 3. A flat delegation system with 3 client-server threads. The foreground client makes a request to itself. The background client makes a request to the foreground server.

how the same hardware thread works as both client and server. Each thread must make progress on its client's work while balancing availability to serve the requests which it has been assigned.

Like Dedicated Delegation, the programmer makes calls to **Delegate_Async**. However, the number of client-server pairs in flat delegation is

$$N_{threads}^2$$

compared to

$$N_{clients} * N_{servers}$$

for dedicated. Maintaining the pending request queues for each client-server pair easily exhausts the L1 cache. Instead, requests are written directly to the server's request line.

To directly write the request line the client keeps track of the most recently used request line on a server. Upon a **Delegate_Async** call, the next response line is handled if available. After handling the response, if the request line is free the request is written out. An unavailable request line implies that this client is waiting on another server. The thread remains productive by serving all of its request lines and then returning to the client. The client will continue invoking the server until its desired request line becomes available.

Like dedicated delegation, **Delegate_Async_Barrier** is invoked to ensure that all outstanding requests and responses are handled before joining.

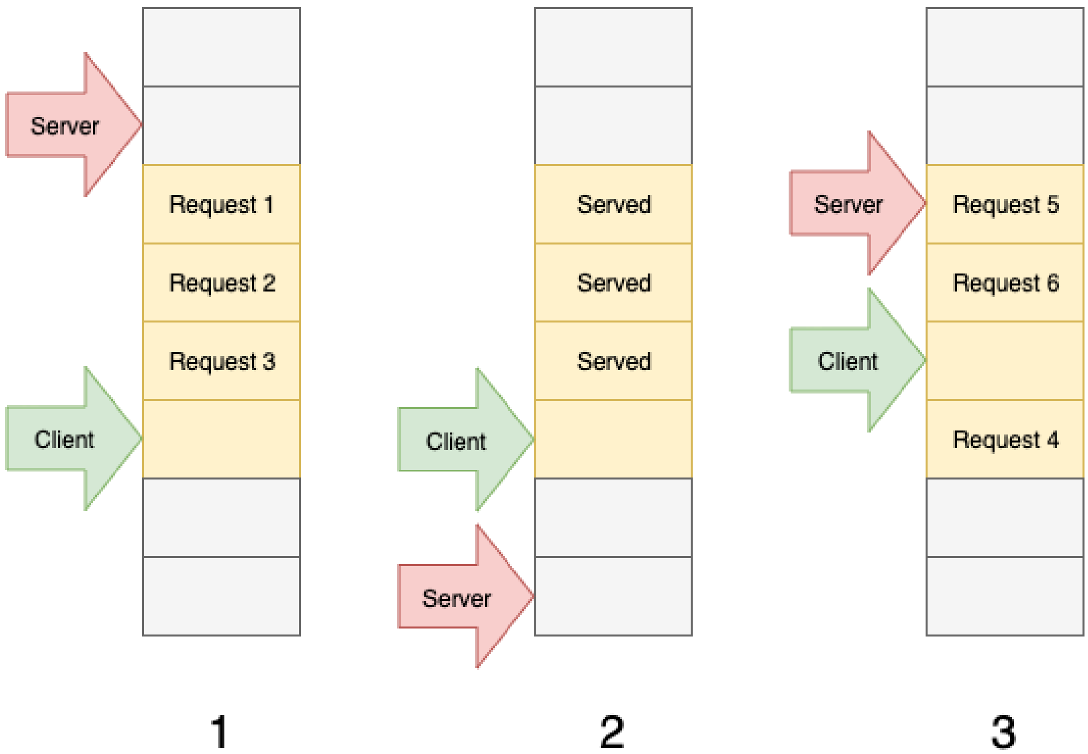

Figure 4. The server may execute requests out of order when multiple requests lines are used by one client. (1.) The client issues 3 requests before the server reaches its section of the request line array. (2.) The server handles all requests in this clients section before the client writes a fourth 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.

2.3 Ordering Guarantees

Beyond the concurrency we can achieve by writing to multiple servers at once, we can also pass multiple requests to the same server simultaneously. To do this we increase the number of request lines available to a client on each server.

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 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, 32kB of L1 cache, 1,024KB of L2 cache, and 19,712kB L3 cache shared among the 14 physical cores on each socket.

For locking, atomic operations, and flat delegation we use the number of threads available on the machine (56) unless stated otherwise. For trials with the dedicated organization we list the number of servers. The balance of remaining threads are clients. The client and server ratio is selected by the best results with 1 variable per server.

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.

Results shown are the weighted arithmetic mean performance over 10 runs of three seconds or greater.

3.1 Fetch and Add Performance

The experiment shown in Figure 5 selects a variable at random and then increments it using the synchronization type shown. Each variable is 64 Bytes and 64 Byte aligned. The locking cases are the posix mutex and spin lock. For the locking case the lock takes the place

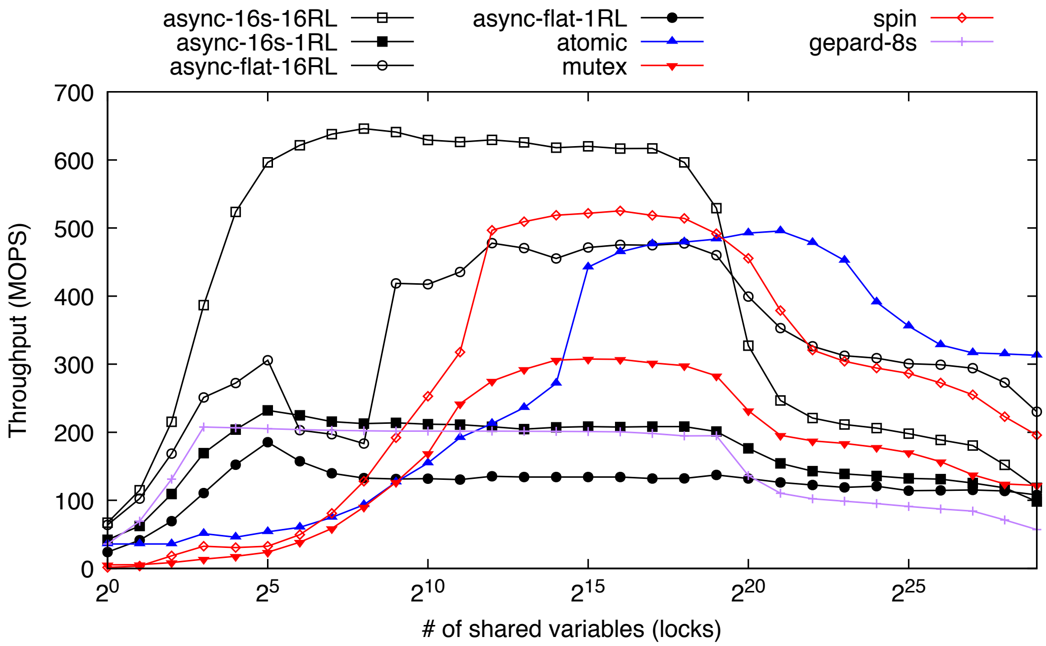

Figure 5. Throughput in MOPS by number of 64Byte variables. Higher is better.

of padding in the variable struct. There is one lock per variable. In cases where variables cannot be delegated evenly, the number of variables is rounded up to the nearest multiple of the number of servers.

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 near 500 MOPS. The reason for this even performance at low levels of shared variables is a lack of contention.

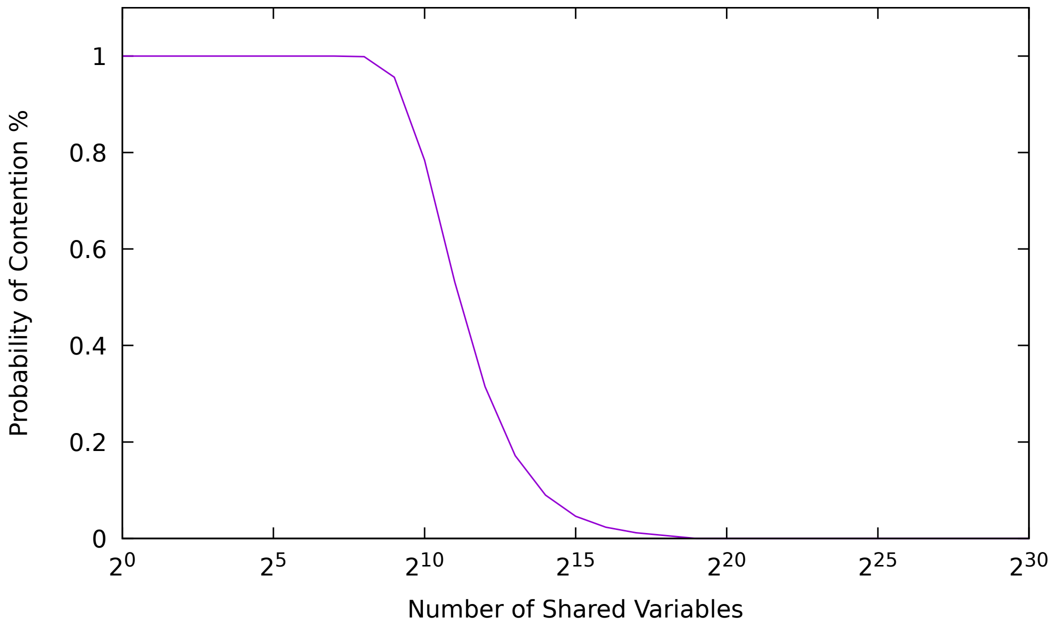

Figure 6. Probability that at least one pair of threads is contending for a variable

A variable is contended if multiple threads are trying to access it simultaneously. Neglecting interleaving due to time, the chance that at least one pair of threads in the system are contending for a variable is plotted in Figure 6 and described as follows. :

$$contention = 1 - \frac{\binom{variables}{threads}}{variables^{threads}}$$

The fine grained locks and the atomic fetch and add have up to 56 threads accessing a single memory location. Figure 5 shows the atomic fetch and add and the fine grained locks reach peak performance as the probability of contention reaches 0 Figure 6. In our microbenchmark, where each thread executes transactions at random, we can assume contention for variables among cores renders the L1 and L2 cache useless because the cache lines are constantly being invalidated by competing cores. We can observe this behavior in the number of L3 lines invalidated by a remote processors L3 cache.

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 k variables per server, or 246 k 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 / $N_{\text{server on processor}}$ of the L3 on the processor, or 2,464 kB. This translates to 39 k additional variables per server, or 630 k 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.

A variable is contended if multiple threads are trying to access it simultaneously. Neglecting interleaving due to time, the chance that at least one pair of threads in the system are contending for a variable is plotted in Figure 6 and described as follows. :

$$contention = 1 - \frac{\binom{variables}{threads}}{variables^{threads}}$$

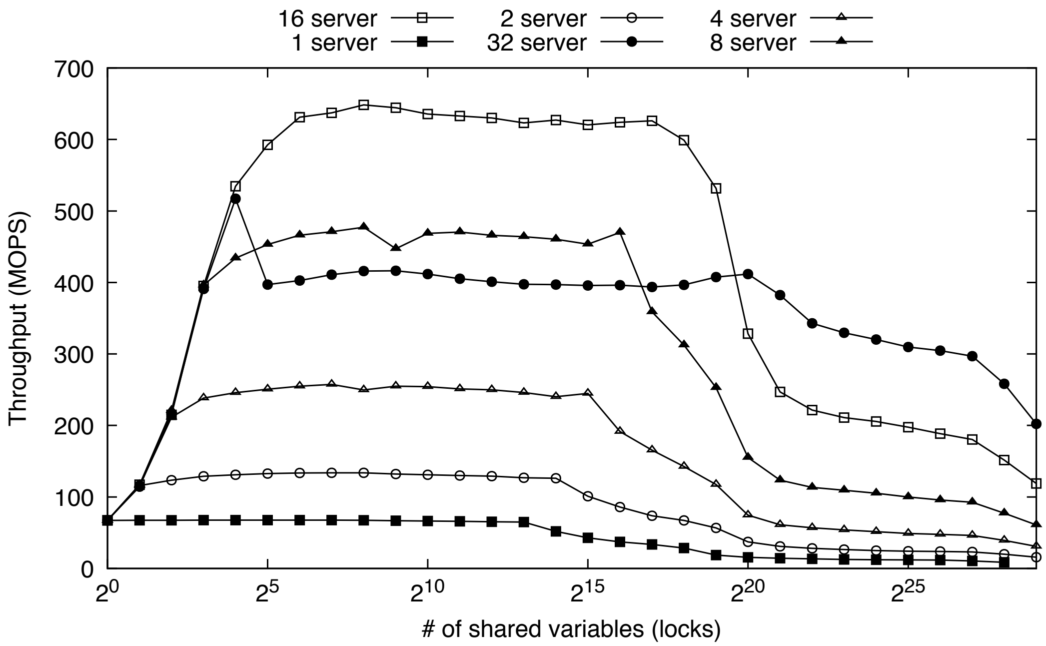

Figure 7. Throughput by Shared Variables by Number of Dedicated Servers

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. 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 7 shows as the number of server cores increases the tail performance of dedicated delegation improves.

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 spin lock or compare and swap will have a 1:$\mathrm{N}_{\text{NUMA Nodes}}$ probability of picking a variable at random that is addressed on the same node as the core. A FIGURE I HAVENT MADE YET shows the number of accesses to remote dram per operation by approach.

$$P_{InCache} = \frac{L2BytesAvailable}{sizeofsharedvariables/numberofservers}$$

As the probability that a variable is in cache approaches zero the time to access DRAM dominates the performance of the delegation server. Flat delegation offers an improvement to other delegation designs in this region because it maintains the same number of threads making memory accesses while ensuring that all DRAM access are local to the thread's NUMA node.

Where atomic operations are unfeasible, we would expect flat delegation to outperform all other options for the extreme end of the range.

3.2 Design Parameters for Dedicated Delegation

The key 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. Ideally the throughput of the system would follow the equation:

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

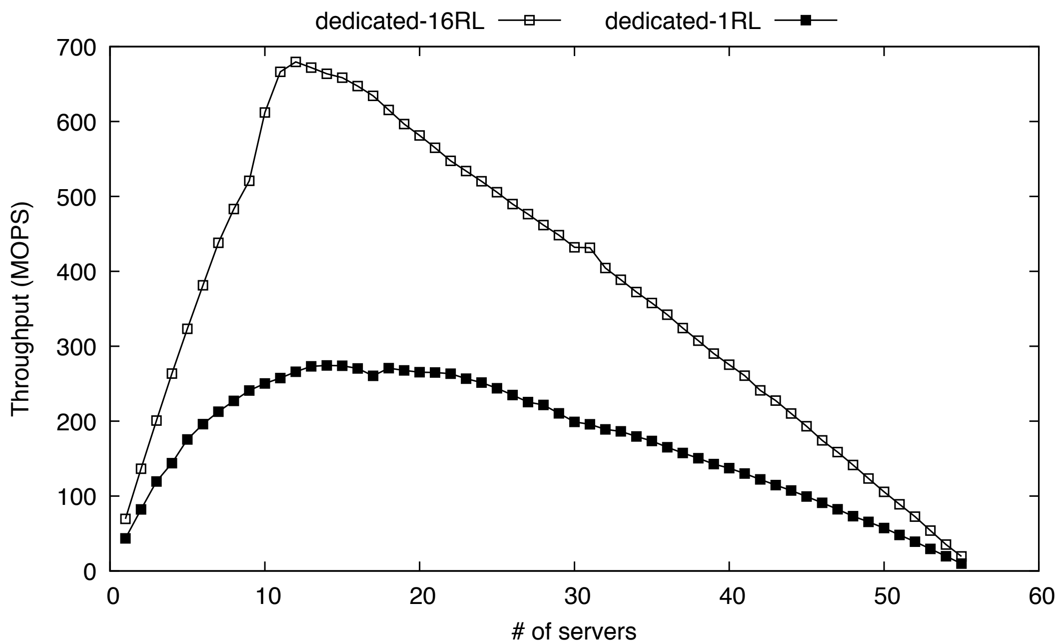

Figure 8. Throughput in MOPS for a 56 thread system by number of servers. The remainder of threads are clients.

The 16 request line case in **??** exhibits the expected behavior. Figure 8 shows the throughput for a simple fetch and add benchmark by increasing number of servers. In the region of the positive slope the system bottleneck is the server capacity. Each additional server added increases the system throughput by the individual server consumption rate. Conversely the region of the negative slope indicates the system bottleneck to be client production capacity. After peak throughput is achieved, reassignment of a client to server duty reduces the system throughput by the individual client production rate. The throughput peaks when the combined server consumption rate matches the combined production rate of the clients.

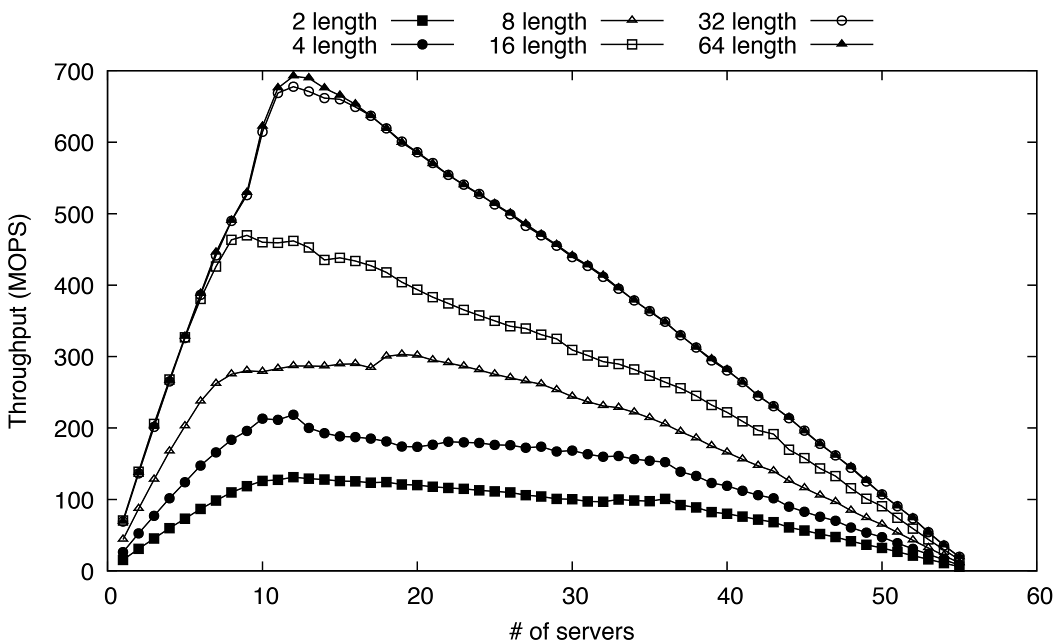

Figure 9. Throughput in MOPS for a 56 thread system by number of servers by pending request queue length. The remainder of threads are clients.

The dedicated design stores requests in an array of local queues, one for each server in the system (see Figure 2). When one of the queues is full, the client writes as many requests as it can. FIGURE shows the effect of the queue length on the fetch and add microbenchmark.

As the queue length increases, the system achieves greater throughput. This is because the probability of all queues having any requests present increases as the queue length increases. When servers are picked at random, the probability of stopping to flush the queues when only one queue contains requests is as follows:

$$P(FullQueue) = \frac{N_{queues}}{{N_{queues}}^{L_{queue}}}$$

This is because the longer queue lengths allow more requests to accumulate in each server queue before the requests are written out to the servers. Writing more requests at once brings greater efficienct to the client, as evidenced by the shallower slope of the ...

Figure: max throughput by client-server combination for all cores case Discussion production rate of clients, consumption rate of clients Figure: Throughput by number of request lines Figure: Throughput by size of ring buffer, no ring buffer Server and Client hit rate

3.3 Design Parameters for Flat Delegation

Instead of specifying the

Figure: Throughput by schedule type Server and client hit rate

CHAPTER 4

CONCLUSIONS

Flat provides and easier API, no tuning. Dedicated allows the user to adjust the producer / consumer rates Applications are programs with smallish data structures with operations more complicated than compare and swap.

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