SCALABILITY BY DESIGN – CODING FOR SYSTEMS WITH LARGE CPU COUNTS

- **Richard Smith**
  - Staff Engineer
- Sun Microsystems

This presentation is based on my experiences over the last 5 months as a member of the mysql performance team headed up by Allan Packer at Sun, and also my longtime interests in High Performance Computing and Computer System Modelling.

Why Care?

- Multi-core multi-thread architectures becoming ubiquitous
  - UltraSPARC T2 @ 64 Threads
  - Derivative designs @ 128, 256 Threads
  - M-series @ 256 Threads
  - AMD Opteron @ 32 Threads
  - Intel Dunnington, Nehalem @ 12—16+ Threads
- Need scalability for multi-threaded programs to exploit them effectively

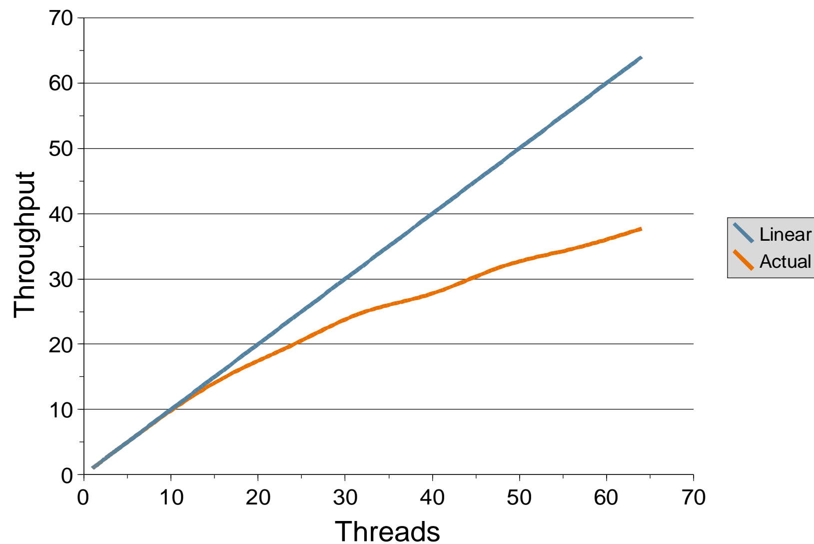
The main reason for caring about scaiability is that multi-core, multi-thread processors are becoming ubiquitous. Hardware vendors are all moving in this direction. Whereas once 4 HW threads may have been considered “big”, even a single socket now has typically 4 or more threads. Sun's UltraSPARC T2 Plus processor is designed to create systems from 64 to 256 threads, which represents a major jump in scale from traditional MySQL deployments.

These new architectures tend to expose scalability bottlenecks, which prevent the underlying HW from being fully exploited. Under certain circumstances, throughput can actually go backwards as concurrency is increased.

The motivation for this presentation is to share experiences around identifying factors responsible for this behaviour and to discuss what we can do about them at design time.

Sun microsystems

Tablescan Scalability Experiment



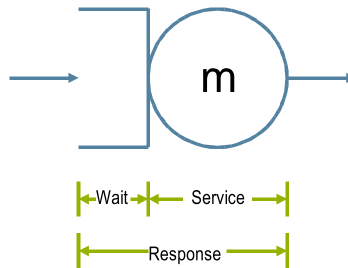
The graph here is based on a simple tablescan workload running on a 64-strand (8-core by 8 thread/core) T5220 CMT (Chip MultiThreading) system. Most of my analysis of MySQL has been on this platform. Super-smack was used to run 1, 2, 4, 8, 16, 24, 32, 40, 48, 56 and 64 concurrent tablescans of the same table. Throughput has been normalised with the throughput at 1 thread equated to 1.

I used a modified MySQL 6.0.6 version for the experiments, one that had several patches applied, and some of my own modifications to reduce the impact of a few known bottlenecks (discussed in a later slide).

The graph shows that throughput does not increase linearly with concurrency, which is typical behaviour. Indeed, at 64 threads the throughput is about 38x. This is still much better than the 4x-8x I've seen quoted on various email aliases. Although not shown, cpu utilisation at 64 threads was about 98%, so at least the HW was being driven reasonably hard with this version of MySQL.

Sun microsystems

Queues



Kendall Notation: $\alpha/\sigma/m/\beta/N/Q$ characterises queue

Arrival, Service time distributions (may be load-dependent)
Number of servers, Buffer & population sizes
Type of service policy: FIFO, RR, PS, SRTF...

It can be useful to model the time spent in a particular place as being associated with a queue. Standard Kendall notation describes a number of common characteristics of a queue, as the arrival distribution, service time distribution, number of servers, buffer size or storage capacity (how many things can be queued up concurrently), and the service policy e.g. FCFS, RR.

Particularly where using hierarchical modelling, the service time distribution might be load-dependent. An example of this is transaction cpu time. In modern processor and system architectures, there are shared resources that can stretch a transaction's cpu time as load is increased. The previous slide reflects this, as it implies cpu time per transaction has increased.

I generally try to be consistent in using the terms Wait, Service, and Response Time as shown above. That is, Response Time is the sum of Wait Time and Service Time.

One special kind of queue is where there are an “infinite” number of servers. In this case there is no queuing: all time spent at the queue is Service Time.

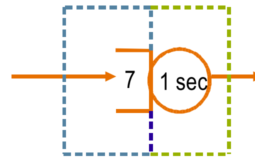
Little's Law

$U = XS$ (Utilisation Law)

$Q = \lambda R$ (Little's Law)

X Throughput
S Service Time
U Utilization

R Average Residence Time
λ Arrival Rate
Q Average Queue Length



7 transactions @ 1 sec ==> 7 secs

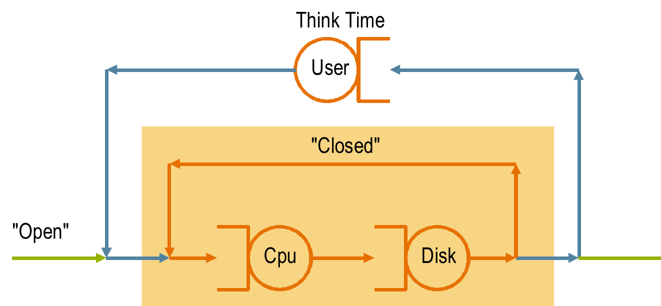
Analysis and interpretation often uses these relationships

5

Under pretty general conditions, a very useful relationship known as Little's Law holds. This relates queue time, queue length and throughput. Intuitively it can be understood as: if there are seven objects in front of me, and it takes on average 1 second to service each of them, then I expect to wait 7 seconds before being serviced.

A specialization of Little's Law is known as the Utilization Law. In this context the “queue length” is simply the fraction of time the server is busy. A resource that is 100% utilized is commonly referred to as saturated and can be a bottleneck. But be careful, this may not always be true as in the case of a HW RAID LUN with multiple physical disks. From an operating system point of view the device may be 100% busy, but it can cope with more work.

Queueing Networks



Can combine submodels into hierarchy of models—FESC
Different levels of detail required
"Response time" is sum of wait and service times at each node
Need both configuration and workload characterisation

6

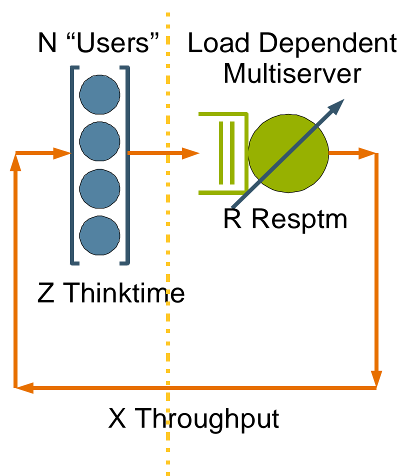
Page 6

The output of one queue can become the input of another. In this way, queuing networks are created. Where the connections form a loop it is known as a Closed network, and where the sources and sinks are external it is an Open network.

A common model of an interactive system combines the cpu-disk loop (also known as central server model), with an outer loop representing users sitting at terminals. In this model, users on average take Z seconds (think time) looking at results of their previous transaction before entering another transaction. The overall user population is usually denoted M.

Little's Law can be applied to a network of queues. In this case the queue length can be interpreted as the population of transactions within a defined region, and throughput is the rate at which transactions leave the region.

High Level MySQL Model



- N users in system
- Thinking or in server
- Load-dependent service time
- Many cpus inside server
- Focus here is on:
  - Cpu-bound threads
  - Minimal i/o
  - "Zero" thinktime
  - Throughput/Scalability

Here then is a possible high-level representation of a MySQL server, in which there are N connected clients. They issue their requests to the Load-Dependent server, and get an answer back on average R seconds later. After "thinking" about the answer, they submit their next query on average Z seconds later.

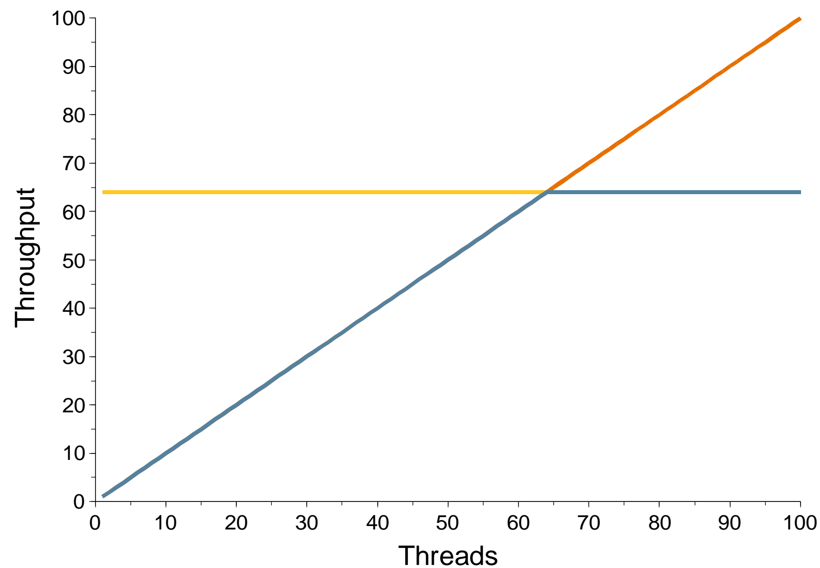
The server has complex load-dependent behaviour, mostly determined by the number of queries to be processed concurrently. Wrapped up in this is contention for resources, both hardware (e.g. cpus) and software (e.g. synchronisation primitives such as locks).

System throughput X and response time R varies as a function of this load, and so does scalability.

Since the focus here is on scalability with large numbers of cpus, we normally try to "short out" the client side by making Z as small as possible, ideally but impractically zero. No network provides infinite bandwidth or zero latency. The Nagle algorithm and deferred Acks can interact badly when using TCP/IP for example.

It would also be a mistake to assume that the load generator in an actual experiment scales perfectly linearly. They can suffer from similar problems to the server e.g. lock contention. All of this is exacerbated when the workload consists of large numbers of lightweight transactions, because modern systems should be delivering sub-millisecond response times for them.

Throughput Bounds

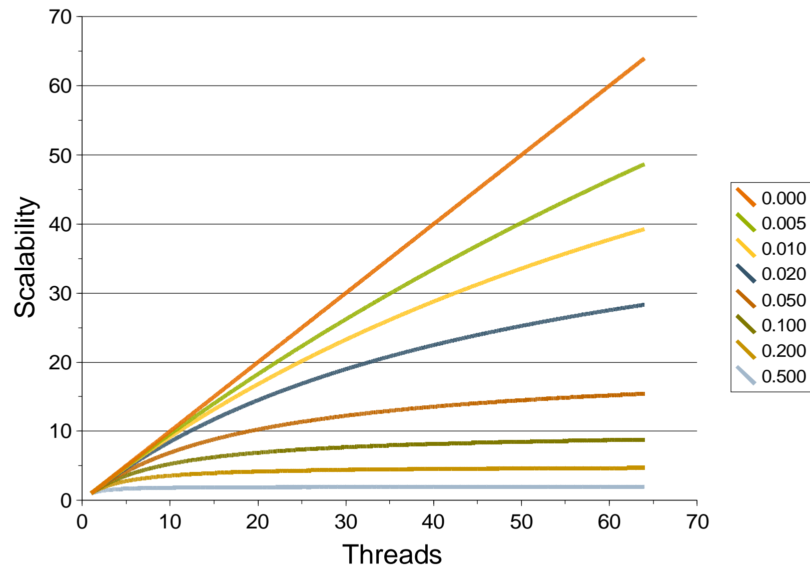


Under an idealised model of system behaviour, system throughput is bounded by at least two curves: the line representing perfect scalability as concurrency is increased, and a horizontal asymptote representing all cpus being 100% busy. More generally, throughput is limited by the “bottleneck” resource, whatever that may happen to be.

In the diagram above, it is assumed each thread is capable of consuming a cpu on a 64-cpu system. Workloads more i/o-bound than that would have a smaller gradient.

Amdahl's “Law”

$$S = \frac{N}{1 - \alpha(N-1)}$$



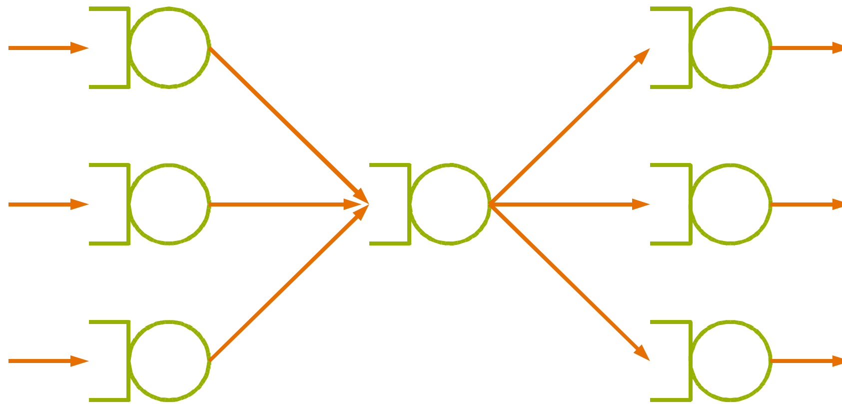
9

A useful, and relatively simple, model of scalability has become known as Amdahl's Law. In the form shown above, it computes scalability S as a function of a parameter α. Larger values of α cause the function to “keel over” sooner and have a lower horizontal asymptote. Scalability in this context isn't a single number, but rather a function, reflecting the relationship between throughput and load.

Comparing this graph with the graph on Slide 3 suggests that Slide 3 corresponds roughly with the curve where α = 0.01.

Sun microsystems

Serial Bottleneck (1)

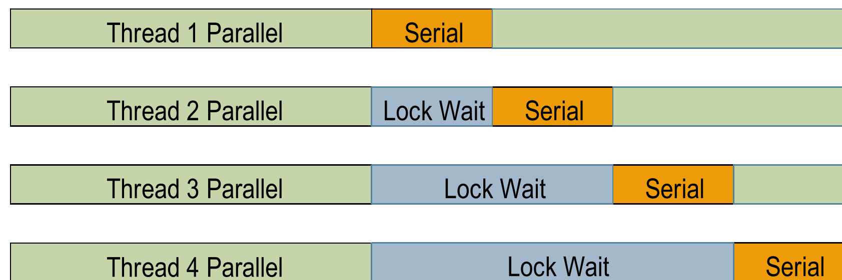


10

Amdahl's Law originally arose from a discussion around parallel computing, in which α represented the fraction of a program that had to be computed serially, versus the fraction (1-α) that could be computed in parallel. In the MySQL context, we can think of it as being the fraction of time we spend executing inside a critical sections, and which at sufficient load tend to become serial bottlenecks. “Bottleneck” here refers to the resource that limits throughput.

Now there will ALWAYS be at least one bottleneck since we don't have infinite resources. When targetting systems with many cpus, we have to be careful that software bottlenecks aren't preventing us from using all available cpus.

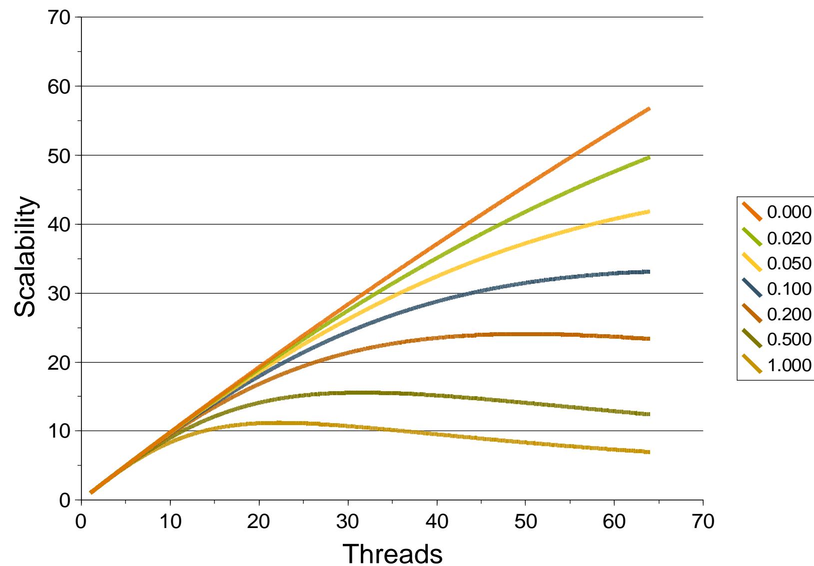
Serial Bottleneck (2)



Solaris: Microstate accounting + profiling, DTrace

At low concurrency, a critical section only introduces a small amount of contention for locks. The longer the time the critical section is occupied though, or as concurrency is increased, a point is reached where the critical section is occupied 100% of the time, and throughput has reached its constrained maximum.

Solaris conveniently measures the amount of time spent by threads waiting for User Locks. On Linux you typically end up using the less direct measure of Involuntary Context Switches. I like to profile mysqld with Sun Studio 12's Performance Analyzer, and explore the callers-callees tree to identify where User Lock time is accumulating.



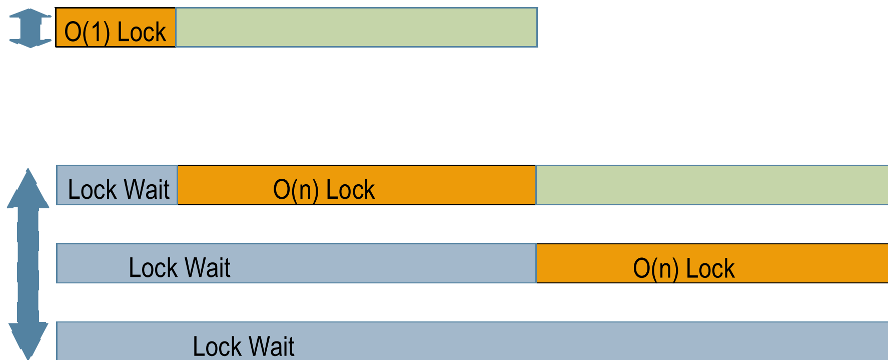
The model underlying Amdahl's Law predicts that throughput reaches a maximum and then stays there as concurrency is further increased. Real experiments however sometimes show a falling off of throughput. A generalisation of Amdahl's Law from Neil Gunther introduces a new parameter β which can generate this behaviour.

In the graph above, when β is sufficiently small and/or concurrency sufficiently low, scalability increases, and can be indisinguishable from Amdahl's Law, or even close to linear. There does come a point though where for some members of the Superserial family of curves, throughput starts to go backwards.

It can be an interesting exercise, given some scalability data, to estimate values for the parameters α and β (e.g. via nonlinear regression), and see what they predict scalability to be like if you should add more cpus.

Sun microsystems

O(N) Serial Bottleneck



Gunther in some of his books refers to α as describing Contention, and β as describing Coherence (the penalty you pay to create a system with coherent memory across many cpus). There is at least one other underlying reason for the behaviour described by β though. It also approximates the behaviour where the time spent inside critical sections increases as load is increased.

Suppose a critical section protects some code that searches a linked list, where the length of the list is O(N), and N is the number of concurrent threads. Since the time spent searching the linked list is O(N), the time spent in the critical section is at least O(N), which means the upper bound on thoughput is being shifted lower and lower. Eventually there will be a crossover where the overall system bottleneck is the critical section rather than the number of cpus available.

The message is that we need to be very wary of critical sections held for long periods of times, and especially of those where the hold time is more than O(1).

Sun microsystems

Case Study #1 Query Cache

- Hash table of queries and previous results
- Single lock protecting entire table
- Hash to list based on query text
- Search list
- Uneven distribution of hashes?
  - Strong Avalanche condition not met?
  - Affects resizing logic
  - Long lists to search
  - 300+ character strings to compare with common prefix
- Hash function character-by-character

14

The Query Cache is an example of a scalability bottleneck in MySQL that can sometimes result in throughput going backwards. For the moment I just turn it off, but I did investigate it to moderate depth a few months ago.

There are several places in MySQL that revolve around HASH tables. Each access to a given table, whether to search, add or delete a record, is protected by an associated lock. In the case of the Query Cache a significant amount of work is done while holding the lock. One concern is that the hash function itself is computed while holding the lock. There is also evidence to suggest that the hash function used is less than perfect, resulting in a poor distribution of records, and lengthy chains of aliases that have to be searched. When searching, lengthy strings (e.g. 300+ characters) are compared, that may differ only in the last few positions. The hash function itself computes things character-by-character. All-in-all, scalability is poor.

Some possibilities for improvement:

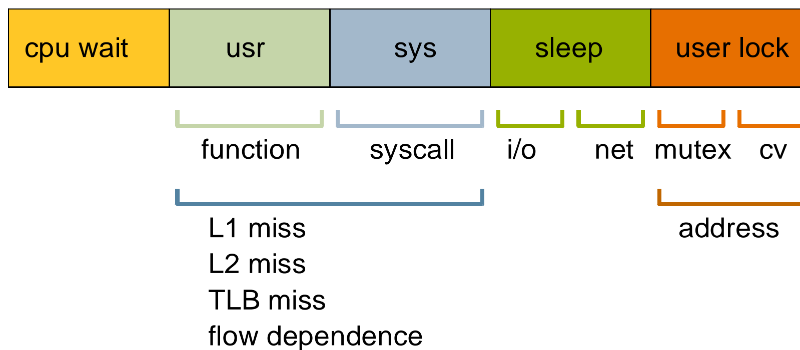
•Move hash function outside of critical section.

•Treat key as array of 8-byte integers and use hash function that satisfies Strong Avalanche condition.

•Store hash values with “key” and compare them first before doing character-by-character comparison.

•Use a Concurrent Hash Table implementation

Microstates



A useful approach in dealing with scalability issues is to break down response times into various categories, such as Solaris' microstates. Using tools such as Performance Analyzer and/or HW performance counters, each microstate can further be subdivided. Having established a reasonable set of categories, then corresponding “states” can be compared between a 1-thread and N-thread experiment.

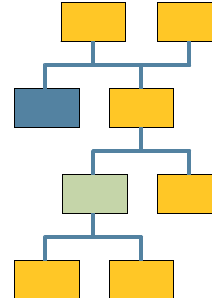
While the focus so far in the presentation has been User Lock time, there may well be other contributors to a loss of scalability. Hopefully by measuring where the time is going, we can stay focused on the biggest bottlenecks, and work on them first.

In the example breakdown above, *i/o* is intended to represent database reads, whereas *net* refers to communication with the client. User lock wait time can be divided into time spent waiting for mutexes vs condition variables [and RW locks and perhaps more exotic synchronisation primitives].

HW counters can help give an idea of the use being made of hardware resources, and perhaps identify reasons for pipeline stalls. This is useful both in single-thread tuning and in reducing interference between multiple threads.

Callers-Callees User Lock Time

Attributed	Inclusive	Function
3416.69	3416.69	log_reserve_and_write_fast
2787.62	3875.38	trx_commit_off_kernel
1565.71	1691.32	trx_undo_assign_undo
340.81	340.81	log_reserve_and_open
308.2	717.9	log_write_up_to
134.85	2011.71	rw_lock_s_lock_spin
60.44	60.44	lock_sec_rec_modify_check_and_lock
59.16	1652.68	buf_page_get_gen
...		
2.57	3877.93	trx_commit_for_mysql
...		
0	2845	trx_undo_report_row_operation
0	8824.29	*mutex_enter_func
8824.29	8840.36	mutex_test_and_set



16

Page 16

The above slide is extracted from a profile of a sysbench workload running at relatively high concurrency (32 threads?) on T5220. It shows the functions responsible for the majority of User Lock time accumulated by the threads processing queries. Since MySQL has a number of background threads, many of which spend most of their time asleep on condition variables, I normally filter them out of these aggregate reports.

One subtlety around this sort of profiling is that tail-call optimisation can sometimes confuse the logic that does stack unwinding each time a sample is taken. When his becomes too confusing, I sometimes compile a special build with lower optimisation, one in which most inlining and tail-call optimisation is suppressed.

By getting source code or disassembler listings of chosen functions, the objects associated with the User Lock time can be identified. In this case its {kernel_mutex, log_sys->mutex, log_sys->os_flush_event}.

In the list above, the function of interest is highlighted in green and has a leading asterisk. The Attributed column shows the amount of User Lock time accumulated by mutex_enter_func and its children when called from each parent function.

Lock Strategy (1)

- Course-grained vs Fine-grained
- Lock overhead
  - contended vs uncontended case
  - spinning vs waiting (adaptive mutex)
- Deadlock
- Livelock and fairness
- Mutex, RW-Lock, other
- Preemption control

17

Hopefully its not a big surprise that issues around Locks dominate much of this presentation. Software that has evolved over many years, particularly when targetting systems with low core-count, tend not to scale well because they use relatively course-grained locking. Identifying these locks may not be hard, but as an outsider it can be very difficult to work out how to break the big locks up into smaller pieces. Some parts of MySQL (e.g. InnoDB) also has certain locks accessible and being manipulated across multiple modules, representing many thousands of lines of code. This represents a major challenge, and suggests to me that over time a process of refactoring is required.

Locks have overheads, even in the uncontended case. This covers the use of atomic instructions implementing functionality like Compare-And-Swap (CAS). In the contended case, a decision has to be made when an attempt to obtain a lock is unsuccessful. Because there is a certain amount of overhead associated with context-switching, it is often desirable to reattempt acquiring the lock before finally getting put to sleep after repeated failures. At a POSIX code level, there is a choice around which particular mutex function to use, and the attributes used in defining a mutex object. By default, pthread_mutex_lock on Solaris is adaptive, whereas on Linux you need to use a non-standard attribute.

Fine-grained locks can create a deadlock condition if they're not acquired in a consistent hierarchical sequence. In complex nests of locks, it can go beyond human ability to analyse whether deadlock is possible. Testing can only demonstrate the presence of bugs, never disprove their existence. An alternative approach is to model the logic and a tool to do the checking. An example of such a tool is SPIN and its Promela language.

While holding a lock is often a really bad time to get preempted. This is increasingly likely if operating with over-committed cpus (more cpu-bound threads than cpus). It can be beneficial to use OS primitives where available to give hints when not to be preempted—on Solaris, schedctl API for example.

Lock Strategy (2)

- MT-safe ≠ MT-hot
  - malloc implementations
- Thread-local data
- Stack semantics (alloca)
- Application-specific memory allocators
- Measure lock impact!

In a multithreaded environment, we generally need all functions to be MT-safe, but preferably also want them to be MT-hot (scalable). Malloc is a frequently and widely used function, but some implementations are not MT-hot. Even with relatively scalable implementations, malloc can consume nontrivial resources. It can be helpful to make use of thread-local data and/or stack semantics (such as provided by alloca) where possible. It can be worth looking at how many times certain functions are called per transaction, such as malloc, and pthread_mutex_lock.

There may be times when some sort of application-specific memory allocator would be beneficial, but premature optimisation is not a good idea either. There are parts of MySQL where the comments indicate a decision was made over a decade ago based on HW that is 15+ years old! I prefer to use native platform facilities unless compelling evidence is collected that they are inadequate. In the case of spin locks, InnoDB doesn't know if a thread holding a lock is currently executing (on cpu) or not, whereas Solaris adaptive mutexes check this condition.

The key suggestion here is to measure lock impact before making a radical decision.

Sun microsystems

Lock Strategy (3)

- Verify Lock Strategy with SPIN/Promela
- Multi-Version Concurrency Control (MVCC)
- Read-Copy-Update (RCU)
- Lock-free algorithms (sophisticated CAS use)
  - Concurrent hash tables
  - Scalable Non-Zero Indicator (SNZI) variants
- Transactional Memory

19

There are some possible alternatives to traditional lock-base approaches:

Use some form of MVCC to increase concurrency of read+write access to shared resources.

Where a pointer is read-mostly, and only rarely used for update, specialised algorithms such as RCU can be considered.

In some cases Lock-free algorithms (typically based on sophisticated use of CAS) can be used to implement highly concurrent access while avoiding the possibility of deadlock. For example, there are algorithms for lock-free concurrent hash tables. [Depending on the specific requirements, maybe a finer-grained lock strategy, similar to java's CurrentHashMap, would work well.]

Recent research into the efficient implementation of hybrid Transactional Memory has as a by-product created some interesting algorithms based on SNZI. SNZI has the potential to scale better than RW Locks for read-mostly access.

Sun microsystems

Key Challenge: Programmability

- Programming multithreaded systems is complex
  - > Need to create critical sections for modifications to shared data structures
- Standard approach: Locks
  - > Expensive
  - > Lock granularity tradeoff -> contention vs. complexity
  - > Scalability -> contention, pessimism
  - > Complexity -> deadlock avoidance
  - > Not composable

20

Despite all the talk about locks, they do have some disadvantages, particularly in large-scale sotware engineering, several of which we encounter regularly in working on MySQL scalability. Considerable research is taking place at present into Transactional Memory and how it might help.

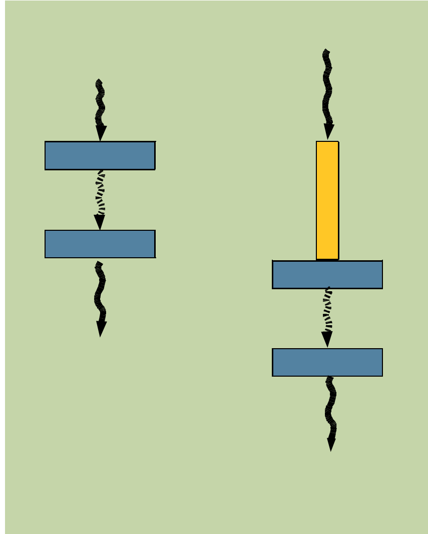
The reference to “Not composable” covers a situation where you are trying to combine two objects, each with their own internal locking logic, into a larger structure. An example might be two List objects where you want to move an element from one list to another atomically.

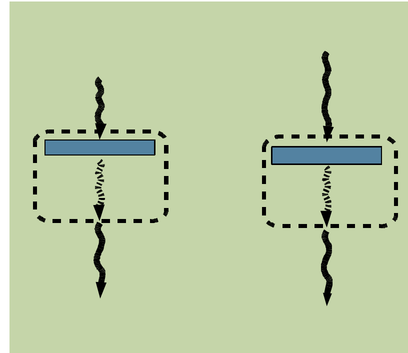
New Approach: Transactional Memory

- Herlihy and Moss, 1993
- A transaction is a finite sequence of machine instructions, executed by a single process, that satisfies two properties:
  - Serializability
  - Atomicity
- Multiple transactions can execute simultaneously as long as their data accesses are non-interfering
- Useful for supporting promising new programming models, and improving performance of existing code

In one sense, Transactional Memory has some similarities with MVCC. Multiple transactions can proceed optimistically BUT if there is an access conflict then at least one of them will need to be aborted. Programmatically, it is desirable to know the reason why a transaction was aborted, and make an informed decsion about how to handle it—whether to repeat the attempt or drop back to a lock-based approach for example.

Eliminating lock bottlenecks with HTM





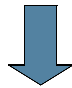
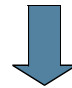
- Use HTM to execute critical section *without acquiring lock*

The traditional lock-based approach requires each thread to incur the overhead of acquiring a lock (which in turn involves one or more relatively expensive CAS instructions to modify shared memory).

Hardware Transactional Memory uses processor features to detect conflict between two concurrent threads and choose one of them to be aborted. There is no slowdown if there is no conflict.

Rock's HTM feature

- Best-effort HTM can abort transactions that exceed resources or encounter difficult events or instructions:
  - Rock examples: function calls, `sdivx` instruction, exceptions
- "Generic" best-effort HTM + Rock extension
  (as in ASPLOS 2006 paper)

```
chkpt <fail_addr>
commit
```

```
rd %cps, <dest_reg>
```

23

One reason for Sun's interest in Transactional Memory is that the capability is being designed into its future Rock processor.

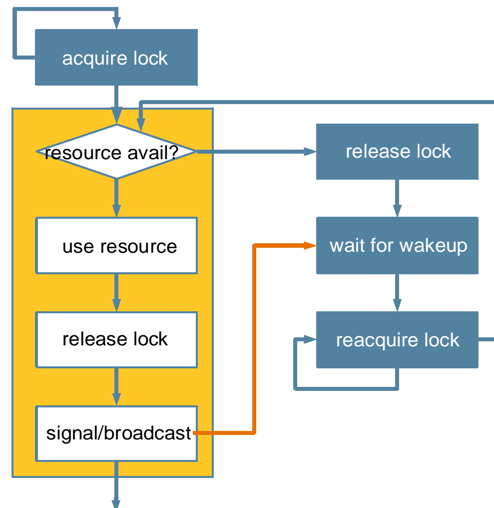
For people really interested in exploring the capability sooner than that, there is a simulator downloadable from University of Wisconsin, based on GEMS.

Preparing for TM

- Hardware TM can abort transaction if resources exceeded—punt to software implementation?
- Keep atomic sections small
- Future language features?
- Active research area
  - Collaborate with SunLabs Scalable Synchronisation Research Group

Based on the information available, its likely that the first implementations of HW Transactional Memory will have a number of limitations. If a transaction attempts to use too many hardware resources then it will fail. Function calls and using large numbers of cache lines are both things to avoid inside transactions. This may be another motivation for moving towards finer-grained locking as an interim step.

Condition Variable



- contend for mutex
- broadcast can cause "thundering herds" problem
- resource may still be unavailable
- repeat...

It would be a Really Bad idea for a thread to wait (sleep) for an event while holding a lock. To avoid this a Condition Variable is normally used. As a byproduct of waiting on a condition variable, a mutex protecting access to the condition variable is released. At some future point in time another thread can wake up the sleeping thread(s) on that condition variable by using pthread_cond_signal or pthread_cond_broadcast. Signal wakes up just one thread, whereas broadcast wakes up all threads asleep on that condition variable. On wakeup, each thread attempts to reaquire the mutex protecting the critical section (this is done automatically inside pthread_cond_wait prior to returning), and control returns is successful. The condition being waited for should be retested on return.

There is a potential problem with using Broadcast, called "Thundering Herds". On wakeup each thread tries to acquire the same mutex, so most are likely to fail, and have to repeat the context switch cycle. The overhead has been so great that at times it has rendered a T5220 I've been testing on unusable, despite putting a shell into RT (Real-Time) dispatch class at high priority.

While its logically safer to wake up everything via broadcast than wake up just one thread, it can have diabolical performance consequences. [It would however also be diabolical to fail to signal a process when required.]

Case Study #2 sync_array

- Major bottleneck
- Implements mutex via condition variables
- Single lock protecting allocation of cells
- pthread_cond_broadcast ==> “thundering herds”
- Killed 64-thread test performance on T5220

```
#define mutex_direct_enter(MX) \
  pthread_mutex_lock(&((MX)->os_fast_mutex))
mutex_direct_enter(&(buf_pool->mutex));
mutex_direct_enter(&(rseg->mutex));
```

26

The use of sync_array in InnoDB is problematic, and is almost guaranteed to create a “thundering herds” problem at high concurrency. It implements mutexes via condition variables, which it allocates from an array, which it protects via a single mutex and holds while a free cell is searched for in the array. Considerable extra overhead is encountered and increases the odds of contention.

For this reason, the version of MySQL referred to in Slide 3 has been modified to use pthread_mutex_lock directly for certain “hot” mutexes rather than use the sync_array mechanism. Its likely that additional locks should similarly be modified.

Hardware Resource Contention

- Competition for functional units
- Shared caches ==> interference
- cacheline sharing, false-sharing, ping-pong
- Non-pipelined long latency instructions
- System interconnect and cache coherency
- Queues for bus access
- XBAR concurrency
- Shared bandwidth to memory
- Local vs remote memory

27

Over time, as lock bottlenecks have been improved, the concurrent tablescan scalability described in Slide 3 has reached a point where it can drive all 64 virtual cpus close to 100% busy. Locks are now no longer the dominant reason for scalability of 38x at 64 threads, and so the focus shifts to hardware resource contention. This tends to be much more processor- and system-specific in the details and magnitude, but are still generic in their general nature. CMT processors by design seek to increase efficiency by multiplexing several threads on the same hardware so they do tend to encounter hardware resource contention.

One way of thinking of a processor is as a highly-specialised message-passing application. Messages can be passed from one part of the processor to another, provided there's a spare buffer receive the message, and a spare slot in the transmission channel to transfer it from source to destination. In a pipeline, if a stage is blocked from forward progress, then it very quickly applies back pressure that block previous stages.

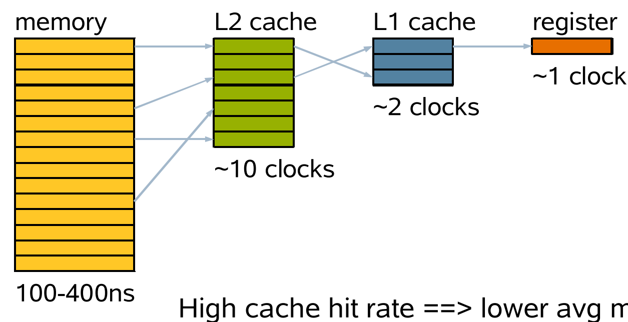
Since memory is so much slower than modern processors a hierarchy of caches is typically used to exploit temporal and spacial locality in memory references. When multiple threads share a cache, then there is always the possibility that they can interfere with each other, so that each experiences a drop in average cache hit rate.

Nothing provides infinite bandwidth, so if the aggregate demand of a shared transmission channel approaches capacity, then queuing should be expected and latency will rise, just as it does at the level of MySQL queries.

Sometimes I like to write artificial code fragments that are designed to stress some aspect of a processor or system architecture, in order to measure what its capable of, then compare the results with the actual usage of real workloads.

Locality of Reference

- Spacial locality: group memory into cache lines
- Temporal locality: keep recently accessed data in high-speed memory



High cache hit rate ==> lower avg memory latency

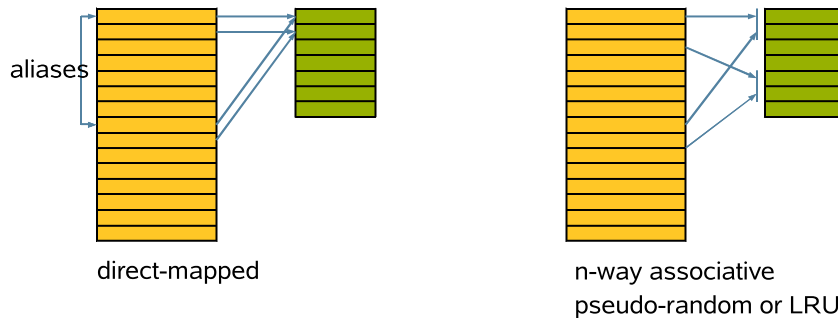
Many codes actually have considerable locality of reference, both spacial and temporal. Spacial locality means that if a memory location is accessed, then its very likely an adjacent memory location will also be accessed. Temporal locality means that it is likely to be accessed again fairly soon.

Caching this data in high speed memory (SRAM rather than DRAM, and possibly on-chip) average memory latency can be reduced. SRAM consumes a lot more power than DRAM and uses more transistors, so the amount that can fit on a die is heavily constrained. For this reason, an external cache (not on the processor die) is commonly used to augment whatever cache is put on the processor chip.

The larger caches usually take several more clock cycles to access than the smaller caches, which means they could stall the pipeline. Part of the challenge in high quality code generation is to mask as much memory latency as possible, including cache latency, and maximise cache reuse.

There is usually more aggregate bandwidth available for caches closer to processors.

Cache Associativity



Hard to do fully-associative caches beyond 64-way
Cache thrashing through aliasing can severely hurt performance

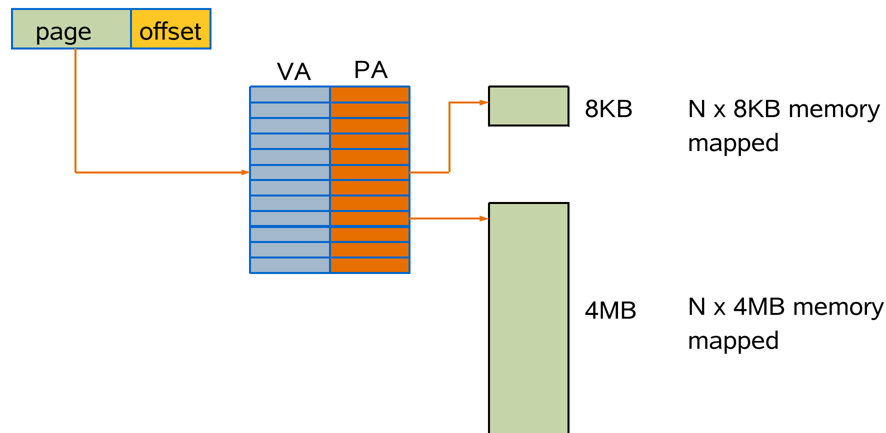
Caches use hashing functions to map a subset of a larger memory into a smaller one. The simplest mapping is direct mapping, where each cache line maps to one and only one location. The downside is that if a fragment of code is working pairwise through arrays that map to the same location(s), they will flush each other out of cache, leading to very low cache hit and poor performance.

An improvement on the direct map scheme is to map to a set of locations and then use either pseudo-random or LRU algorithm to select a location within the set. The width of the set is its associativity. Small caches e.g. 64 entries tend to be fully associative, whereas larger caches are around 2 to 8 way associative.

On processors that support it, Performance Analyzer can be used to profile an application based on its memory objects, not just on code. For memory-related HW counters, it periodically records not only the Program Counter, but also Virtual and Physical addresses of the memory reference responsible for triggering an interrupt.

The resulting Dataspace profile can be datamined, looking for unevenness in the distribution of requests to resources.

Large pages and DTLBs
(Data Translation Lookaside Buffers)



Any memory reference involves a virtual address (VA), which may need to be translated into a physical address(PA). Special hardware, known as a Translation Lookaside Buffer or TLB, is used to contain an active subset of virtual to physical mapping. Entries in a TLB correspond to a page, since all virtual addresses within a page will have the same physical page address.

As with any other cache, its size will be constrained, which means that when an address to a new page is first looked up a DTLB miss will occur. An entry will have to be chosen to be flushed and the new entry inserted in its place. This may take many clock cycles depending on whether its done by hardware or a trap to an operating system routine.

One way to reduce the impact of DTLB misses is to use larger pages where supported. A table of 512 entries can map only 4MB when using 8KB pages, whereas the same TLB could map 2GB when using 4MB pages.

Processors are now starting to support even larger pages e.g. 256MB and 1GB. Solaris supports multiple pagesizes, and provides mechanisms to allocate appropriate sizes automatically if it can. There is also an API to specify explicitly the sizes desired. On Linux this is less integrated, and requires explicit use of hugetlbfs (a relatively low-level wrapper around the logic for loading DTLB entries).

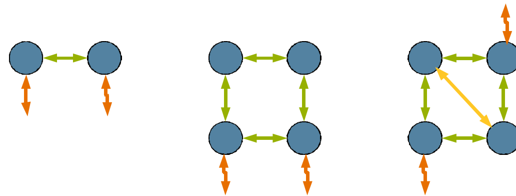
Hardware Performance Counters

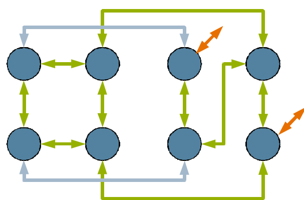
- Program them to count events of interest
- Understand interactions of code with pipeline
- System-level
- Per-LWP (virtualised counters)
- cpustat(1M), cputrack(1), libcpc(3LIB), ...

HW counters are highly non-portable, but most processors have them. They can be very informative when tuning both single- and multi-threaded code. Some examples appear in subsequent slides.

Sun microsystems

NUMA Architecture (AMD: SUMA)





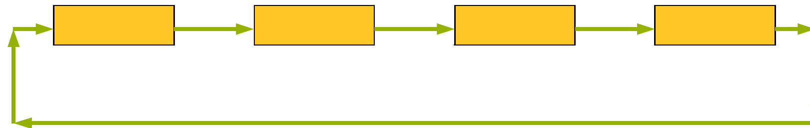
Each hop adds 30 – 40ns latency
and consumes system bandwidth.
Consider memory placement carefully.
libnuma, madvise...

32

Page 32

Its a major challenge to build very large SMP systems with completely uniform memory latency. These days, its common to use some degree of non-uniformity (NUMA) in order to scale. In these architectures, local memory has lower latency than remote memory. There may also be more bandwidth available to local bandwidth than remote memory. For workloads that are memory-bound, performance may be able to be improved if a way of maximising local vs remote memory references can be found. This makes sense for thread-local data, since threads tend to have some degree of cpu-affinity. However for large shared structures such as a database buffer cache, it may be more appropriate to ensure its physical memory is evenly distributed across the system. APIs such as libnuma (Linux) and madvise (Solaris) can be used to give hints to the OS about physical memory allocation.

Memory latency across hierarchy



- Pointer-chasing benchmark
- Huge range of behaviours by varying:
  - Size of ring (L1, L2, Memory regimes)
  - Stride between pointers
  - Use of prefetch (software, hardware)
- Many codes stall heavily on memory

33

A particularly interesting workload to explore involves reading around a ring of pointers. By varying the size of the ring and the stride between consecutive pointers, L1, L2 and main memory latency can be measured. Many codes stall heavily on memory, so although a pointer-chasing benchmark is fairly extreme, its useful for understanding which counters describe what is happening in the memory hierarchy.

X4600M2 Memory Latency

ring size	32k	512k	512m
tsc	3,002,696	16,735,114	328,306,576
bu_cpu_clk_unhalted	3,003,018	16,863,401	328,373,667
fr_retired_instructions	1,250,850	1,254,125	1,342,789
fr_retired_branches	125,172	125,865	144,506
fr_retired_taken_branches	125,150	125,852	140,256
fr_dispatch_stall_ls_full	2,500,589	16,260,516	327,239,502
dc_accesses	1,000,415	1,002,084	1,042,605
dc_misses		1,000,253	1,005,083
dc_refills_from_l2		1,000,178	
dc_refills_from_system_exclusive			1,001,882
bu_writeback_l1_to_l2		1,000,293	1,006,567
bu_system_read_responses_exclusive			1,002,147
nb_cache_block_read			1,003,592
nb_cpu_mem_from_local_to_local			1,004,279
nb_cpu_mem_from_local_to_remote			

X46000M2 has a relatively NUMA memory architecture, and there is a significant difference in memory latency depending on which pair of cpus are involved (home node for the memory and execution node accessing that memory). The experiments listed here all involved local memory. Even local memory latency varies though across the 8 sockets, due to the way the cache coherence protocol is implemented and the time taken to propagate cache probes across the system and receive responses.

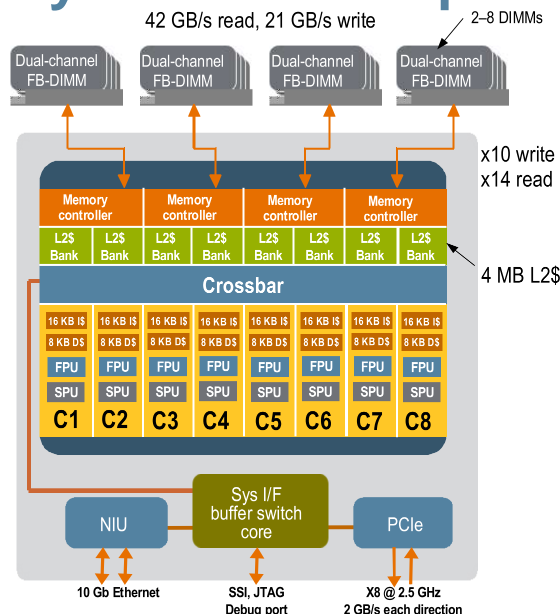
Firstly, it can be seen that *tsc* and *bu_cpu_clk_unhalted* are roughly measuring the same thing, the number of clock cycles that the program ran for. When the pipeline stalled, it was mostly due to the load-store buffer being full with outstanding requests for data *fr_dispatch_stall_ls_full*. Loads and misses to L1 cache are counted in *dc_accesses* and *dc_misses*. When L1 is missed, then the cacheline has to be loaded from either L2 cache *dc_refills_from_l2* or from memory, *dc_refills_from_system_exclusive* in this case. A cacheline gets evicted from L1 to make room for the new cacheline *bu_writeback_l1_to_l2*.

The 512M case shows 1M reads coming from memory *bu_system_read_responses_exclusive* and *nb_cache_block_read*. The requested data were all in local memory *nb_cpu_mem_from_local_to local* rather than remote memory *nb_cpu_mem_from_local_to_remote*. These are synthetic events created by combining *NB_cpu_io_to_mem_io* event with unit masks 0xa8 and 0x98.

Sun microsystems

UltraSPARC T2: System on a Chip

- 8 SPARC V9 cores @ 1.2–1.4GHz
- 8 threads/core
- 2 integer execution pipes/core
- 1 Load-Store Unit/core
- 8KB 4-way L1$/core
- 4 MB 16-way 8-bank L2$



35

In the UltraSPARC T2 processor architecture (which is at the heart of T5220), each core has its own L1 data cache. On the other side of a Crossbar is an L2 data cache, shared by all cores. The L2 is divided into 8 banks, and each bank is responsible for accessing and maintaining a portion of physical memory. The bank to be accessed is determined by physical address bits bits 8..6. This means that memory references rotate around the banks every 64 bytes of PA space, and after 512 bytes returns to the same bank.

The trap here, which often crops up in High Performance Computing, is that we tend to allocate data structures that are powers of two. If not careful, a hot spot could be created in which one bank is accessed much more often than all other banks. The Crossbar can only transfer one request per bank per cycle from the 8 cores. A hot spot increases the possibility of queuing, which in turn would increase average L2 access latency.

Sun microsystems

Concurrent 2GB Tablescans

Threads	1	8	16	32	64
Cycles	66,665	67,652	72,018	86,479	111,175
Instr_cnt	19,749	19,770	19,729	19,702	19,783
Instr_ld	3,354	3,375	3,368	3,362	3,368
Instr_st	1,192	1,220	1,221	1,221	1,209
Atomics	11	12	12	12	12
Br_completed	4,353	4,405	4,345	4,347	4,367
Br_taken	3,411	3,490	3,455	3,457	3,449
DC_miss	232	282	460	885	1,339
L2_dmiss_ld	23	19	20	18	20
CPU_ld_to_PCX	364	424	592	1,013	1,473
CPU_st_to_PCX	1,280	1,307	1,309	1,310	1,297
Idle_strands	37,206	36,221	26,766	15,299	4,638

All numbers are in Millions

Since I was interested in learning more about possible HW reasons for 38x scalability at 64 threads, I collected a set of HW counters on T5220, and a subset are shown above. The numbers represent averages per thread. Highlighted in orange are the significant changes when moving from 1 to 64 threads. There has been a drop in L1 cache hit rate (DC_miss has increased), and a corresponding increase in the number of load requests being sent to the L2 cache (CPU_ld_to_PCX).

Originally I couldn't reconcile the increase in clock cycles with the moderate increase of accesses to L2 cache. The data implies an average L2 latency at 64 threads of around 40 cycles, which is a substantial increase from 23 cycles in the uncontended case. A future experiment would be to collect a Dataspace profile and observe whether there is an imbalance in L2 Bank accesses.

Sun microsystems

Gratuitous Advice

- Use fine-grained locking or lock-free strategy
- Document the strategy, including correctness criteria (invariants)
- Keep critical sections short
- Profile the code at both light and heavy load
- Collect HW performance counter data
- Identify bottleneck resource (there's always at least one!)
- Eliminate or ameliorate it

At this point sheer exhaustion and thirst brings me to a close, with the following gratuitous advice in order to scale well on large numbers of cpus:

SCALABILITY BY DESIGN – CODING FOR SYSTEMS WITH LARGE CPU COUNTS

•**Richard Smith**
–richard.smith@sun.com

For those interested in corresponding further on any matters raised, here's my email address. Please note that I'm based in Melbourne, Australia, and that there can be a rather large timezone difference between me and much of the rest of the world.