Cache Effects:

An Analysis of Cache Behavior on Multi-Core Hardware

Loren McGinnis and Benjamin Zagorsky

December 14, 2011

Introduction

This project began as a set of measurements on the validity of a new scheme for scheduling many-threaded processes on multi-core hardware to achieve better cache hit-rates. While the planned scheme was ultimately not supported by the data, we found several interesting things in the process. This paper contains a plethora of performance benchmarks on multi-core hardware.

Some highlights of analysis include an assumption of the default Linux scheduler and an approximate number on when code should be written in parallel versus serial. The default Linux scheduler assumes multi-threading processes can be parallelized, even if this is not the case. It turns out that for heavily contentious processes (applications that run serially but were written threaded), the Linux scheduling policy does as bad as possible, by running the threads of the processes on different cores rather than all on the same core. If you are writing threaded code for Linux that isn't actually parallelizable, you should set the affinity of the processes to one core.

As for what code should be parallelized, we determined that if the average probability of a memory access being contended is less than 5%, the code would benefit from parallelism. Otherwise, running it on many cores will only slow it down. This number was shown on a virtualized 8-core system. For 8 hyper-threaded cores (4 physical cores), this number was higher (around 17%), due to cores sharing caches.

Procedure

We ran a suite of benchmarks to test cache behavior on modern multi-core hardware. The benchmarks were all memory strides that interacted with every page (4096 bytes) of memory in some way, either by reading from it, writing to it, or locking and then unlocking a mutex stored on it (both pthread locks and a spinlock implementation). All benchmarks were run under Linux. The benchmarks were run on both an 8 core virtual machine with Centos 6 and on a 4 core hyper-threaded (so 8 virtual core) physical machine running Ubuntu 11.10.

The full system specs are as follows:

VM:
Virtualized CPU 2.2GHz X 8
64-bit
8 GB RAM
L1 cache: 64K
L2 cache: 512K

Physical Machine:
Intel® Core™ i7 CPU Q 720 @ 1.60GHz X 8 (Hyper-threaded)
64-bit
4 GB DDRAM
L1 cache: 32K (shared between pairs of virtual CPUs)
L2 cache: 256K (shared between pairs of virtual CPUs)
L3 cache: 6144K (shared between all CPUs)

The benchmarks varied in the number of pages they strode over the powers of two from 2 to 2^{15}, inclusive. To normalize the data, the number of times the benchmark strode was set so that each run would touch the same total of 2^{27} pages (so for 2 pages, it would touch each page 2^{26} times, while 2^{15} would touch each page 2^{12} times). For each stride length, the benchmark was run in eight different modes: two for each of four possible thread configurations.

- Same Core: 8 threads with affinity on the same core.
- Different Cores: 8 threads all set to run on all different cores.
- No Affinity: 8 threads with no affinity (assigned to cores by the Linux scheduler).
- One Thread: A single thread (does an eighth the total work, or as much work as one of the eight threads in the other modes).

Within each of these modes, the threads would either run on all the same memory or all on different memory.

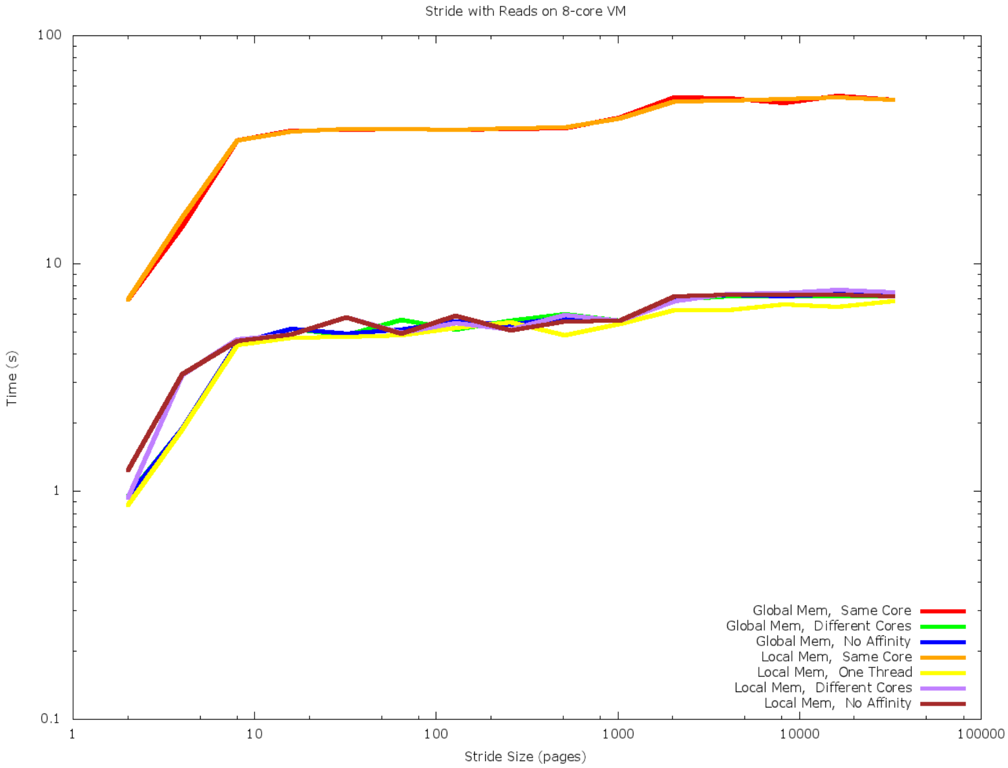
For clarity, threads iterating over the same memory is referred to threads dealing with global memory, while threads iterating over disjoint subsets of memory is referred to as threads dealing with local memory. This is not local in the sense of sitting on a thread's stack; all the memory involved was all allocated on the heap. These terms are just help to disambiguate the memory modes from whether threads running on the same or different cores. As a notational example, "Global Mem, same CPU", or global-same refers to a run all the threads are on the same core and accessing the same memory.

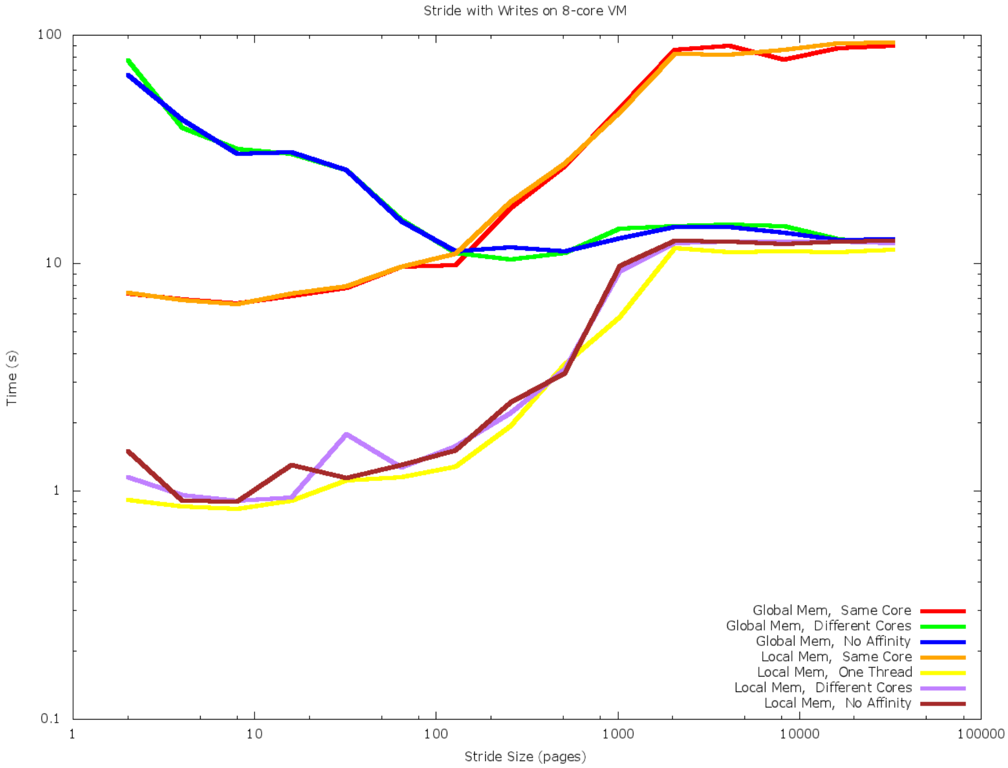
Thus, there were 56 different data sets, each a member of the following cross product (one-thread/local and one-thread/global behave identically, so the latter is omitted):

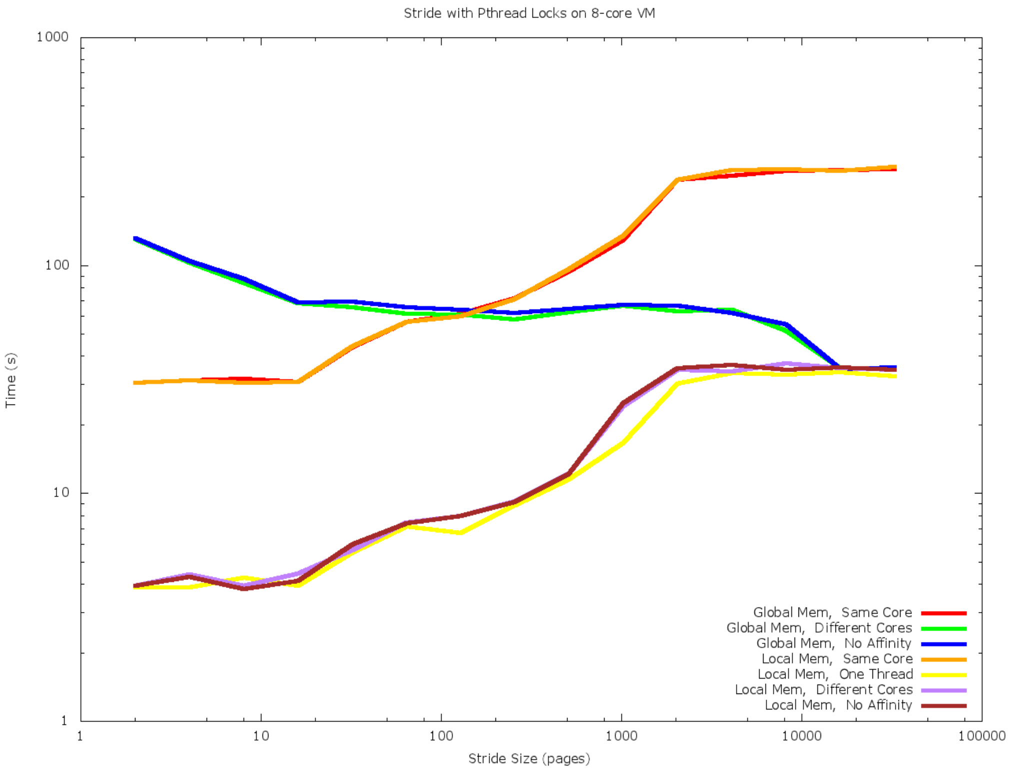
{hyper-threaded, VM}
X
{read, write, plock, spinlock}
X
{same core, different cores, no affinity, one thread}
X
{local memory, global memory}

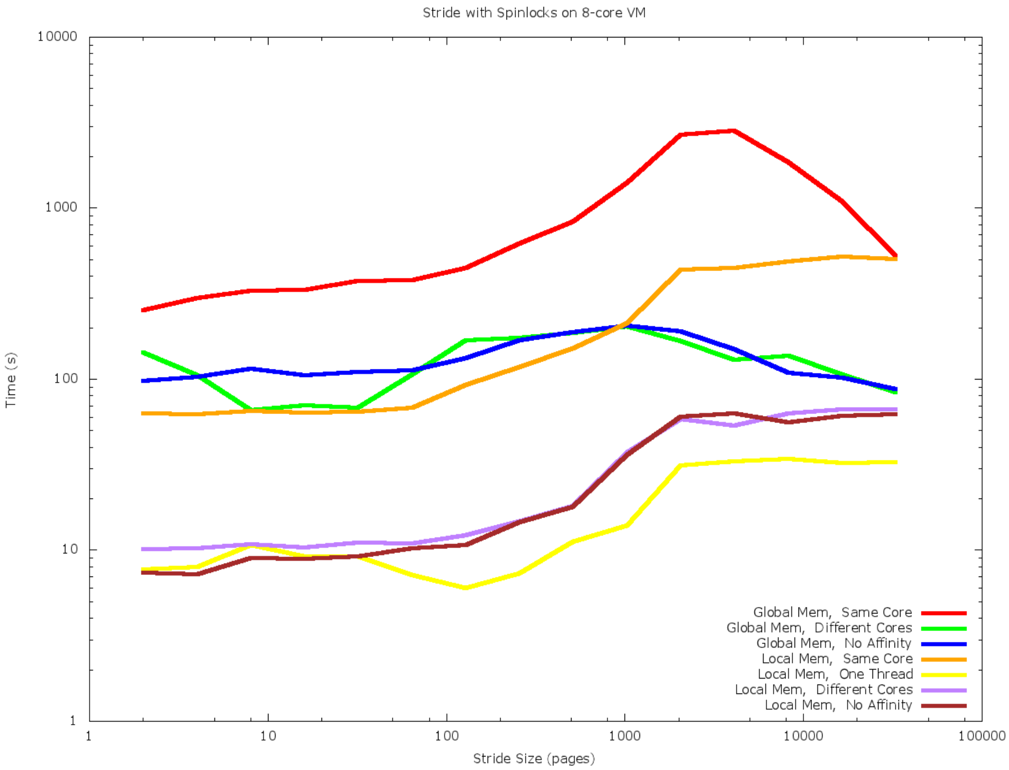
The real time for the runs of all of those cases were measured using the gettimeofday() system call before the threads started running and after they rejoined the main thread. The following 8 graphs are the run times for reads, writes, pthread locks, and spinlocks on the regular and hyper-threaded cores.

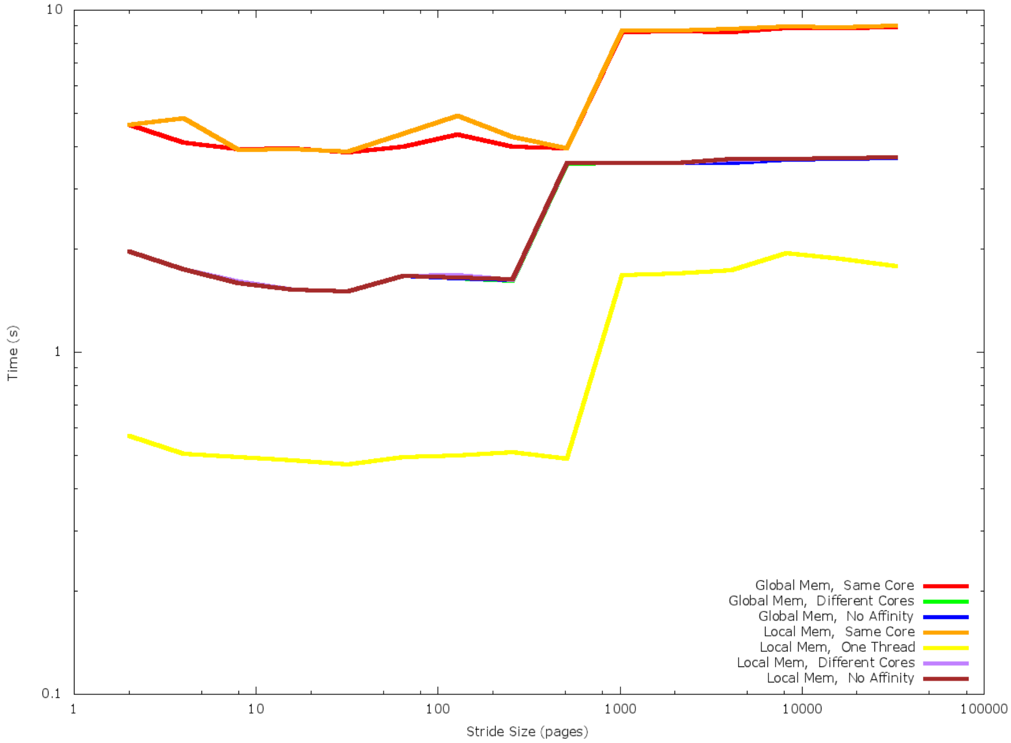
Results


Stride with Reads on 8-core VM
Time (s)
100
10
1
0.1
1
10
100
1000
10000
100000
Stride Size (pages)
Global Mem, Same Core
Global Mem, Different Cores
Global Mem, No Affinity
Local Mem, Same Core
Local Mem, One Thread
Local Mem, Different Cores
Local Mem, No Affinity

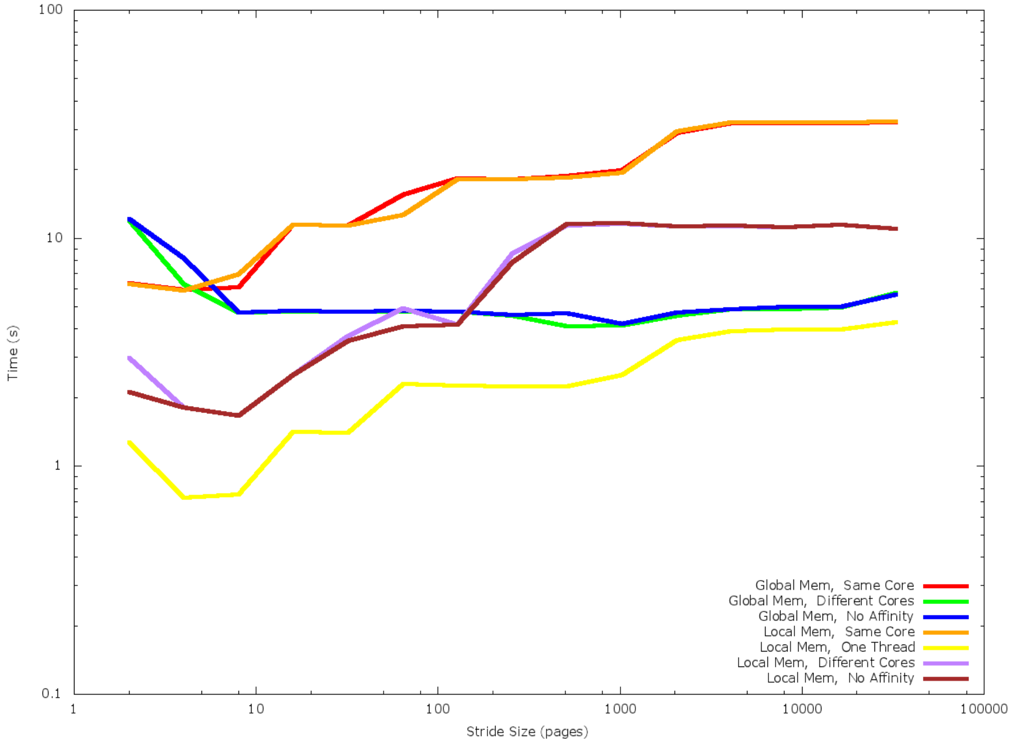

Stride with Writes on 8-core VM
Time (s)
100
10
1
0.1
1
10
100
1000
10000
100000
Stride Size (pages)
Global Mem, Same Core
Global Mem, Different Cores
Global Mem, No Affinity
Local Mem, Same Core
Local Mem, One Thread
Local Mem, Different Cores
Local Mem, No Affinity


Stride with Pthread Locks on 8-core VM
1000
100
10
1
Time (s)
1
10
100
1000
10000
100000
Stride Size (pages)
Global Mem, Same Core
Global Mem, Different Cores
Global Mem, No Affinity
Local Mem, Same Core
Local Mem, One Thread
Local Mem, Different Cores
Local Mem, No Affinity

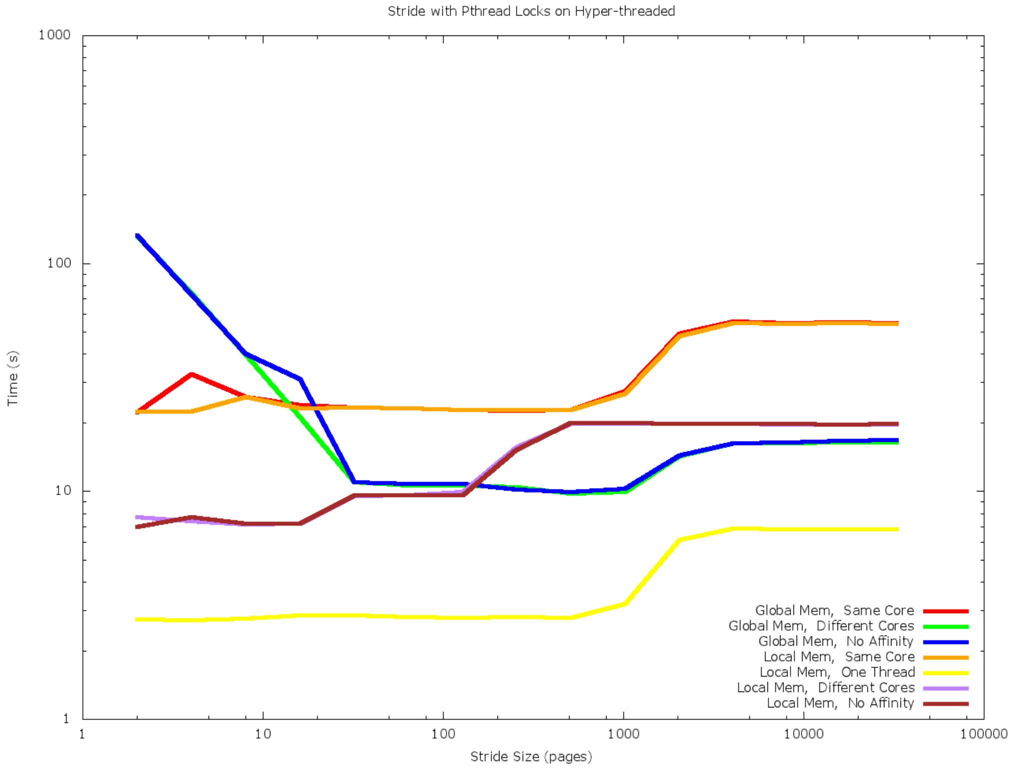

Stride with Spinlocks on 8-core VM
10000
1000
100
10
1
Time (s)
1
10
100
1000
10000
100000
Stride Size (pages)
Global Mem, Same Core
Global Mem, Different Cores
Global Mem, No Affinity
Local Mem, Same Core
Local Mem, One Thread
Local Mem, Different Cores
Local Mem, No Affinity

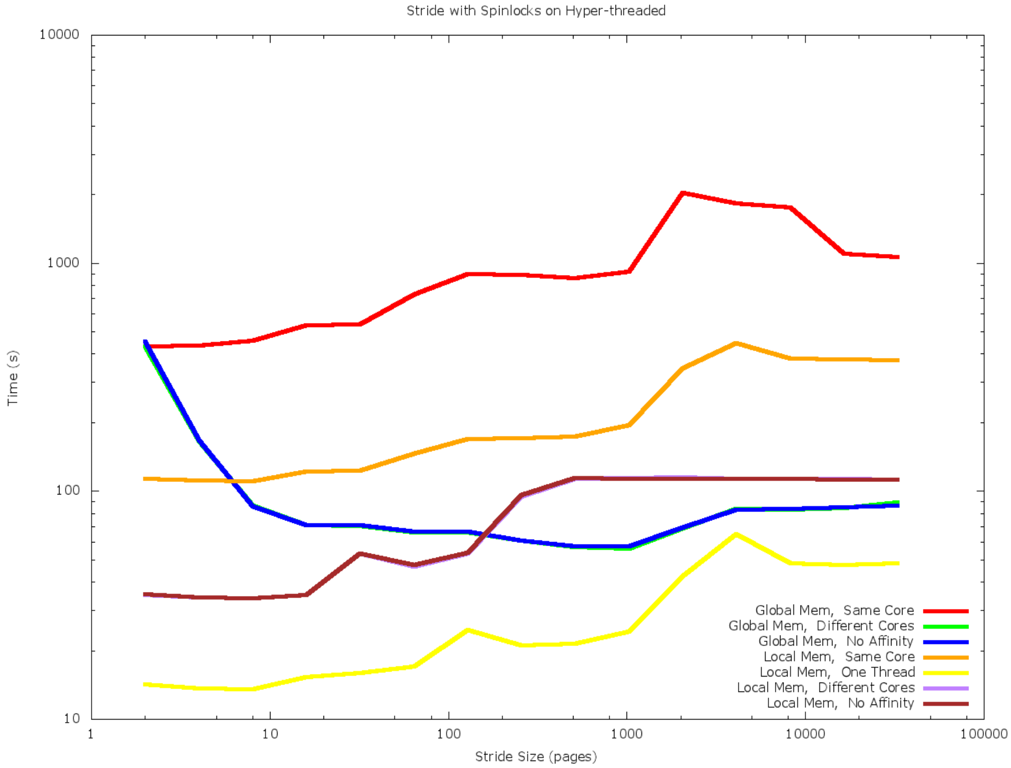


Stride with Reads on Hyper-threaded



Stride with Writes on Hyper-threaded





Note that for ease of viewing, all graphs are log-log plots

Since the data points are normalized so that each one represents the same amount of work done per thread, we would expect in the absence of any memory caches that all of the lines would be horizontal. In that case, each thread is doing only raw memory accesses, and the costs of cache coherency are absent. Thus, any deviation from a straight line is due solely to the presence of caches. In every benchmark on both machines, there is a usually a jump in run time indicating the exhaustion of the L1 cache. For the VM, this occurs between 1024 and 2048 pages; for the hyper-threaded machine, between 512 and 1024 pages. This exactly corresponds to the number of cache-lines in each respective cache: 1024 and 512. Thus, all the memory accesses fit in the cache for the lower number, and at least half of the memory accesses for the higher number are always cache-misses. The poster-child example of this is one-thread, which exhibits this behavior reliably. However, there are a few exceptions to this rule, for which the explanations are forthcoming.

A pair of lines that behave identically (except for spinlocks) is global-same and local-same. This is true across both sets of hardware, implying that any possible performance benefit from the threads all sharing a cache is negligible compared to the run time of this particular application. If this benefit were significant, then the global-same would have lower run time than the local-same, since, in the global case, one thread accessing a piece of memory would mean it was cached by the time another thread accessed it. We are wont to conclude that on the smaller memory strides, the number of memory accesses that are saved are insignificant compared to the number of accesses that are made. Meanwhile on the larger memory strides, the entire stride doesn't fit into cache anyways, and so the cache benefit of this locality is mitigated by evictions. With spinlocks, when running on the same CPU and accessing global memory (local memory has no contention; it serves as a baseline), the CPU spins on itself, wasting an entire timeslice every time it hits a locked spinlock. This is extremely inefficient, and results in global-same doing worse than everything for every stride size. Only when the likelihood of contention becomes very small with larger stride-lengths (above 4096 pages), does same-core global approach the behavior of same-core local.

Another significant phenomenon is that the one-thread benchmarks perform identically to the local-different benchmarks on the 8 core hardware, but not on the hyper-threaded 4 cores. On the hyper-threaded hardware, the one-thread benchmarks outperform the local-different benchmarks by a factor of two in most cases. There is one section in all hyper-threaded benchmarks, between 128 and 512 pages, where the run time for the local-different increases, while the one-thread does not increase until after 1024. This indicates that hyper-threading is not beneficial for parallelized data access. This is due to the fact that each pair of virtual cores than run on a physical core must share the same cache, which counters the benefit of having two execution contexts on a single physical core. Essentially, the cache-size is halved for each physical core when threads are running in both contexts.

Analysis

An interesting ratio is that between the local-same and local-diff runs. For writes this is 6.96, for reads it is 7.13, and for locks it is 7.36. The expected ratio would be 8, a linear scale, but moving the threads to different CPUs, we discovered, causes greater variation between thread run times. With eight threads running, there will be variance in how long each takes to run, even with them doing the same thing. The program will only finish when all eight threads are done, which means the local-diff run is the run time of the slowest thread. Meanwhile, the local-same run is eight times the run time of the average thread, rather than the slowest one. This conclusion is supported by the "getrusage" system call (which was run in preliminary data collections), which reports the total time spent by threads of the process running as well as the wall clock time. This demonstrates a rather significant variance in the run time of the threads; potentially as much as 25% between the fastest and slowest, even when they are doing identical work on identical hardware. The reason this variance exists between different cores, but not on a single core, is because of the heterogeneity of memory access between cores. The arrays that each core access were allocated just with malloc calls all at once. There is no control over what portions of physical memory are allocated for each array, so some of the cores may take more or less time to access their assigned memory. Since the total run time of a piece of parallel code is bottlenecked by the slowest thread, the wall-clock run time of an entire process will be skewed upward by this variance. When comparing two datasets that have the same core utilization, this variance is either present or absent and both, and should not invalidate the comparison.

Perhaps the most interesting feature of this data is the difference in performance between global-same and global-diff on both writes and locks. These two runs perform identically on reads, which makes sense because this is a test of concurrency; readers can act in parallel, but writers and locks must be synchronized. Even though the writers were not synchronized software-side, they perform as if full concurrency control is in effect. The reason for this is cache-line bouncing. When one core writes to memory held in the cache of another core, the hardware invalidates the cache of the other core. This is slightly surprising: the failure to synchronize contended memory accesses does not improve asymptotic performance of parallel applications (though there is some constant factor speed up; for example, pthread's locks are slower than raw memory writes by a factor of about 3.9); even if the programmer does not explicitly synchronize writes, hardware with strong consistency models will. It should be noted that locks are memory writes themselves, and so the pthread and spinlock implementations of locks exhibits similar behavior to dumb blind writes. We expect that cache-aware locks like the MSC lock[1] would not suffer from the same cache-bouncing problem.

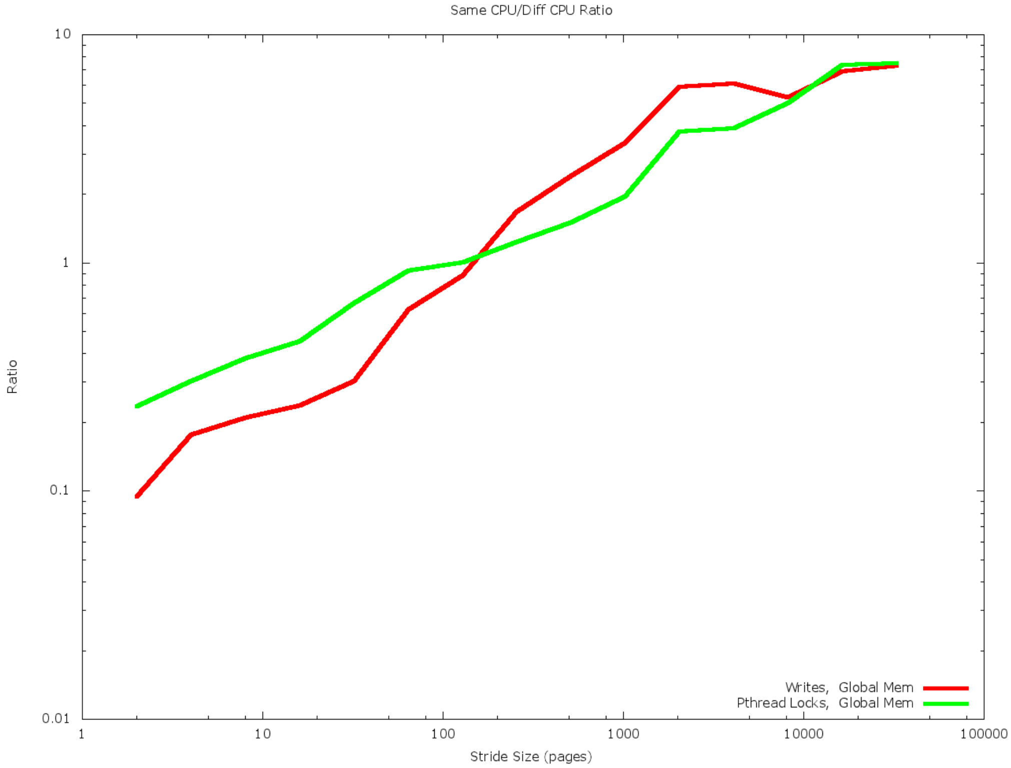
So the act of contending areas of memory with writes inherently slows down code, both software and hardware side. But how does this slowdown relate to the frequency of contention, and what does this say about parallelizability of code? The answer to that lies in the ratio of run times of global-same to global-diff on writes and locks. For a given stride length, if this ratio is less than one, this means that the effectively serialized code (the code all run on one core) is faster than the parallel version; the cost of synchronization is higher than the gains from parallelism. Conversely, if this ratio is greater than one, the parallel version is faster. Meanwhile, the stride length is inversely related to the probability of contention. Since pthread locks and writes behave very similarly, we can speak in terms of locks (note that this does not apply to spinlocks, due to the aforementioned problem that they waste many cycles in the same-core case). At any given time in the thread benchmark, a thread is trying to acquire one, effectively random, lock. The number of total locks is simply the stride length L. There are seven other threads doing the exact same thing. Thus, the probability that at least one other thread is accessing that lock simultaneously is $1 – (1 – (1/L))^7$. This is also the expected fraction of memory accesses that will be contended.

[1] "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", John M. Mellor-Crummey and Michael L. Scott, ACM Trans. on Computer Systems 9(1), Feb. 1991, pp.21-65

Given this, it makes sense that the global-same/global-diff ratio improves as stride-length increases. It reaches 1 when the stride-length is around 128, which corresponds to a contention probability of approximately 5%. An approximate conclusion that can be drawn from this is that an application should only be parallelized if the contention of probability of an average memory access is less than 5%; otherwise the serial code would be faster. However, this is an over-simplification. This will be discussed in detail later. Explicit best-fit formulae for the global-same/global-diff ratio with relation to stride-length are as follows:

$\text{Ratio}_{\text{global-same to global-diff}} = 7.44 – 33.2 * e^{(–0.000176 * L – 1.59)}$
$\text{Ratio}_{\text{global-same to global-diff}} = 7.3 – 32 * e^{(–0.0005 * L – 1.5)}$

The first is computed from the data directly (it is the best fit of formula of the form $f(x) = a + b * e^{(c x + d)}$); the second was hand-tailored to have slightly better accuracy on the lower numbers (2 through 256) and simpler coefficients. One thing this equation calls out is that the data reaches an asymptote for large stride lengths; and that asymptote is the exact same ratio between the run tines of global-diff and local-diff. Essentially, for large stride lengths, the probability of contention is low enough that it is not a factor at all, and the parallel code running on the same memory behaves exactly the same as parallel code running on different memory. So, unsurprisingly, parallel code with a very low chance of contention behaves exactly the same parallel code with no contention at all. On a hyper-threaded architecture the critical stride length for which the parallel code was faster than serial code occurred much lower; around 16. Treating this as a four core system (there are 4 sets of cache), there is a contention probability of 17%.



A graph of the global-same/global-diff ratio for writes and locks. At 128 pages the ratio is 1, where contention is low enough for parallelism. These ratios are from the 8-core VM data.

Evaluation of the Default Linux Scheduler

The default Linux scheduler, represented by the no affinity lines, behaves identically to the cases where all eight threads run on different cores. This is true across all of the different runs on both sets of hardware, implying that the behavior of the default Linux scheduler is to, at least for lack of other things running on the system, run threads of a process on different cores. This is better than running the threads on a single core in all cases except for contentious writes (this being global-no-affinity writes and locks on small stride lengths). As just discussed, these are cases where the code in question should not have been written in a parallel manner. Essentially, Linux assumes the code you are running will benefit from parallelism and schedules accordingly.

At a glance this seems like a good scheduling policy. Users should not be running multi-threading code if it is not parallelizable. However, in the event that they do, should they be penalized for it? Threads can be a nice abstraction as well as a means of parallelizing code. While Linux does require additional information to determine which processes ought to be serialized (have all their threads run on a same core) and which can be parallelized, this information is not much. Linux would only have to count cache shoot-downs per memory write by process, and serialize those processes for which this ratio was above some calibrated threshold.

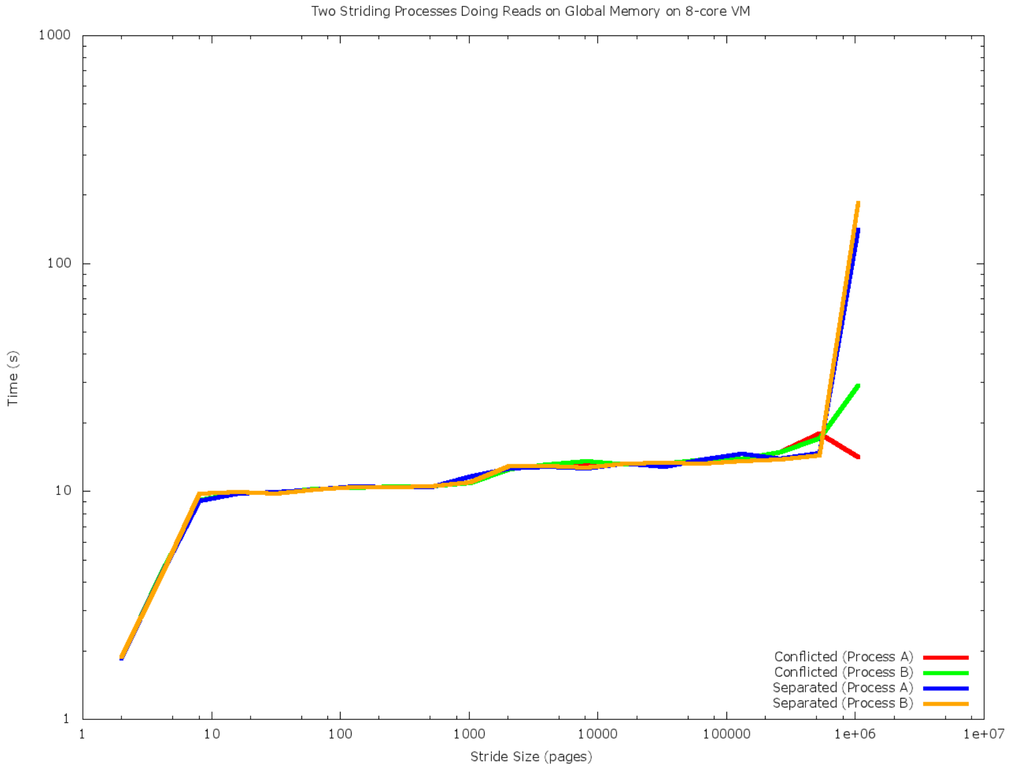
For lack of changes to the Linux scheduler, there are two easy work-arounds to this problem. First, don't use threads in serial or heavily contentious code. This may not be possible for every application, in which case is the second solution: set the affinity of the processes to a single core. Then Linux will only schedule the threads of that process on that core, and, as our data demonstrate, for some applications can vastly improve their running time.

Our Initial Scheduler Concept and Other Challenges

The plan was to minimize the cache flushes incurred by context switches. The scheduler would reserve one or two cores for the kernel and small processes (small in CPU usage) and allocate the remaining cores explicitly to CPU-heavy processes. The threads of one of these large processes would then only be run on a core allocated to that process, and nothing else would run on allocated cores, thus eliminating the need to flush the TLB in between threads of CPU-heavy processes.

There are two problems with this scheme, as indicated by our data. First, running CPU-heavy threads on the same core versus different cores, even when they were accessing the exact same memory, did not generate any significant improvement from warmer caches (see the comparison between the global-same and local-same datasets). Second, the process of reserving cores would potentially cause CPU-idle time, especially on the reserved small process core. This was initially thought to not be a problem, as the better cache performance would more than make up for this idle time. Instead, we observed the exact opposite. For good parallel code, you want threads to run on all available cores. Especially since the cache benefit of running threads on the same core was too small to be observable, the idle time incurred by this scheme would vastly outweigh the benefits.

The following graph shows the results of a benchmark running two processes at the same time:



In the "conflicted", the processes each have a thread on every core (again using affinity). In the "separate" case, one process runs on four cores, the other on the other four. Until the strides approach the amount of RAM on the system, there is no discernible difference between the two cases. For the very large strides, running them separate actually is detrimental to their performance. This is due to paging. When one of the processes has a page fault, potentially all of its threads have to wait for a page to get paged in. In the separate case, this means that the four cores cannot make progress until the page fault returns. In the conflicted case, the process that did not fault can still occupy CPU time. The faults are rare enough that all the cores can be kept busy, but their cost is many orders of magnitude, so the moments when cores are idle in the separate case accounts for the spike in the graph.

One challenge we encountered is that affinity of processes is inherited. This was problematic because on our earlier benchmarks, we set the affinity of all other processes in the system to a reserved core 0 to test this scheme. As a result, all of the runs where we were not setting the affinity of the threads defaulted to having affinity for core 0, which is not at all the behavior of the default Linux scheduler. This caused some quite specious, although exciting, results.

Conclusion and Future Work

Through a thorough analysis of how caches affect the stride program, we have been able to determine the memory contention threshold below which an application should be written in parallel and above which it should be written in serial. We measured this to be 5% on the 8-core hardware, but the much higher 17% on the 4 hyper-threaded cores.

It must first be noted that these numbers are deceptively low. A 5% probability that any two threads both hit the same part of memory is actually an extreme case of contention. However, one can imagine workloads that contain points in memory that are hotly contended. A global counter that is updated often could approach this contention rate. In a database workload where a key or datum receives frequent updates, the performance could be improved by serializing access to that portion of memory. Changing the code to reduce contention is probably the best way to maintain parallelism, but where this is impossible, the code simply needs to be serialized through some means to increase performance.

In the modern multi-core realm, writing code to be multi-threading is often touted as a way of making it run faster. Some programs can be split into completely independent parallel components, and these should obviously be parallelized. Other programs can only execute in serial and these should remain that way. But many programs are somewhere in-between; they can run in parallel but with points of contention. How much contention is too much? 5% is a nice number, but it has already been shown to change in the face of hyper-threading. It may also vary with number of cores and likely varies with memory access speed. Further investigation into the effects of these and other causes could be very interesting and fruitful.