# SMP scaling considered harmful

**Revision 1.4**

*Larry McVoy*

BitMover, Inc.
San Francisco, California
415-821-5758
lm@bitmover.com

*ABSTRACT*

We put forth the controversial idea that scaling can be a harmful thing to a general purpose OS if carried to far. The level of harm is directly proportional to the amount of scaling and is worse than linear in the number of processors. We claim that converting a uniprocessor OS to a 4 way SMP OS introduces only a small amount of damage, but converting the 4 way SMP OS to a 32 way SMP OS does a much larger amount of damage. We call this phenomenon the “locking cliff.”

The point of this paper is not to say that scaling is a bad idea. Running on large machines is a requirement for many users, which means it is not an option to say that we’ll just give up on anything past 4 processors. The real points of the paper are to (a) remind people that scaling comes with a cost, and (b) to set the stage for a follow on paper which describes techniques which can be used to get to larger numbers of processors.

## 1. Outline

- What is scaling?
- Scaling is good, right?
- How is it accomplished?
- Run queues: an example
- The locking cliff
- Costs
- Benefit
- Cost / benefit comparison
- What should we do?

## 2. What is scaling?

When we want to achieve higher performance than is possible from a single CPU, we add more CPUs. In order to use them for anything but user level problems, we need to modify the OS to be multi threaded, that is, allow multiple CPUs in the OS at the same, possibly looking at and/or modifying the same data. These modifications allow all CPUs to work in parallel, even when the work is all in the kernel. For I/O bound work loads, multi threading is a critical performance feature.

### 2.1. Scaling definition

There are many definitions of scaling, here is the one we are using. Assume N CPUs in a balanced system (balanced means there is a reasonable balance between I/O and CPU, for example, a 128 CPU system with one I/O slot is extremely unbalanced). Plug in an even mix of disk and networking I/O devices. Run workloads through the system (web benchmarks are a good choice, they do both network and disk I/O). As long as there are CPU cycles and bus bandwidth available, the system has not reach the scaling limit until all the I/O devices are saturated. In other words, the I/O devices should be the bottleneck, not the CPU devices.

## 3. Scaling is good, right?

Everyone likes the idea of their system being faster than any other. This has been true of everything from horse drawn chariots to race cars, and computer systems are no different.

The world tends to accept without question that scaling is a good thing. After all, faster is better, right? Maybe not. The media and marketing folks certainly use scaling as a marketing tool. Consider the recent Mindcraft NT vs. Linux benchmarks.[1] Leaving aside whether the benchmarks measured anything useful or not, the more interesting point is that for the load generated, instead of using a single 4 way SMP machine, 4 uniprocessor machines with a DNS round robin (or better) load balancing scheme would have solved the same problem with higher performance, and would require no change to the OS to achieve the increased performance.

The point is not to say that high performance is a bad goal, it isn't. The point is to question whether SMP scaling is the only and/or best way to reach that goal. This paper will attempt to show that SMP scaling is not healthy after a very small number of CPUs (around 4), and that there are other ways to achieve high performance.

## 4. How is scaling accomplished?

The basic problem in making an MP system scale is that the OS itself must scale. That means that multiple CPUs will want to be in the OS at the same time, possibly accessing and/or modifying the same data structures. Scaling is the process of allowing this to happen without allowing more than one CPU to modify data at the same. There is a great deal of literature on this, both at the hardware level and the software level. Readers interested in this area might want to read about the various hardware consistency models, they form a good basis for understanding the issues. Hennessy and Patterson's Computer Architecture book is a good starting point.

One may think that the multiple CPU problem is new, but it is not. Even on a uniprocessor, the OS must protect itself from itself. Multiple threads of control can be in the OS at the same due what is known as bottom half/top half concurrency. Device interrupts can cause the OS to execute OS code while there is already a process in the OS processing a system call.

[1] http://www.mindcraft.com/whitepapers/nts4rhlinux.html

### 4.1. Threading the kernel

The process of scaling the operating system is deceptively simple in concept. Start out with one lock around the whole operating system, figure out why it is a contested lock, split it into two locks, each of which protects different sections, and repeat until the system scales.

The goal is to end up with a system which never collides on a lock. It is OK to get there right after some other CPU has released lock, the cost of taking that lock is on the order of a microsecond (a few cache misses at most). It is not so good to have to wait for a lock; that means that the process wanting the lock has to be put to sleep and woken up later when the lock becomes available. That can take 1000 times longer than just getting a lock. It is so expensive, in fact, that many systems use spin locks, where they just repeatedly try for the lock until they get it (smarter systems only spin a few times and then give up; tuning the number of times is hard).

### 4.2. Run queues: an example

Consider the problem of scheduling multiple processes on multiple CPUs. The simple implementation, which works well for some work loads, is to have a queue of waiting processes. When a CPU becomes available, the next process is removed from the queue, context switched onto that CPU, and allowed to run. Since all CPUs will be modifying the run queue, the queue is protected by a lock.

This simplistic implementation actually works reasonably well for certain types of jobs. The perfect sort of job for this implementation would be one which is CPU bound and runs for exactly one time slice and then exits. Jobs which are I/O bound do not fare well because they tend to run for short periods to schedule the I/O and then go to sleep waiting for it to finish. Their ratio of run time to reschedule time is poor, which increases the contention for the run queue lock. Longer running jobs don't fair so well for a different reason, known as the cache affinity problem. A global run queue does not take into consideration the "cache footprint" which a process builds during its timeslice. If a process is switched from one CPU to another CPU, the cache footprint must be rebuilt in the new CPU's cache. A global run queue can do a pretty good job of making the system caches appear to be 1/Nth of their actual size, where N is the number of CPUs. Not good.

To solve the scaling problems in the run queue, most designers will create one run queue per CPU. The scheduler will try hard to put a process on a particular run queue and never move the process between processors. The design leads to better scaling through less lock contention. The same design goals which eliminate lock contention also result in the processes moving around less; this yields more benefits in the form of cache affinity.

In a well-implemented MP scheduler, the simple run queue abstraction is maintained, but there are N instances of it, trying hard not to talk to each other. The goal is to successfully partition the CPUs from one another so that there is a minimal amount of lock contention. The CPUs still have to talk to each other when moving processes from one CPU's queue to another (to load balance), but the goal is to avoid that for two reasons: the cross CPU lock traffic and the loss of the cache footprint that occurs when a process is moved.

## 5. The locking cliff

The locking cliff is the point at which it becomes easier for a programmer to just add another lock rather than determine if there already is a lock covering the data structure in question.

To understand the locking cliff, we need to look at the different kinds of locking models, the level of difficulty of each model from the programmer's point of view, and then consider the typical actions of a reasonable programmer working in each model.

### *5.1. Locking models*

There are a number of different locking models. Even uniprocessor machines have locking issues.[2] The following sections contain a brief review of the most common locking/threading models.

#### *5.1.1. Single lock*

One way to get asymmetric multi processing, which is better than nothing, is to put one big lock around the whole kernel. Everyone takes this lock, whether entering from the top or the bottom. Most uniprocessor operating systems try out this model when they are trying to run on MP hardware. This model results in very little or no scaling of the operating system on MP hardware. User processes, such as compute bound jobs, can scale quite well so long as they stay out of the kernel.

[2] Uniprocessor operating systems use disabling interrupts as their locking model.

#### *5.1.2. Coarse grained locks*

In order to get concurrency in the operating system, the operating system must allow more than one process (or interrupt) to execute in the operating system at the same time. To do this, we divide the OS into sections and give each section a lock. For a small number of processors, we only need a small number of locks, each covering a fairly large region of the OS.

This model of coarse grained locking will provide good scaling on small numbers of processors, but poor scaling on large numbers of processors.

#### *5.1.3. Fine grained locks*

As the number of processors increases, the number of locks must also increase if we wish to scale up the OS. The limit, as the number of processors goes to infinity, is probably about one lock per cache line. In other words, a lot of locks. Solaris, around the 2.1 release, had over 3000 statically allocated locks, with many more allocated dynamically with data structures (i.e., open a file, get another lock, create a socket, get another lock or three).

Fine grained locking can result in near perfect scaling, although it is not without some cost. All those locks add instructions and data. The larger machines benefit from this but the smaller machines pay a price.

### *5.2. Locking requirements for programmers*

Each of the locking models require the programmer to understand the model and obey the model's constraints. The following sections briefly describe what the programmer has to understand and do in order to work within each model.

#### *5.2.1. Single lock*

The single lock is an easy model for the programmer to understand because there is exactly one lock. If you want to be in the kernel, you have to wait until no one else is there. The only thing a programmer needs to do is to remember to take this lock if entering the kernel. Since adding entry points to the kernel is a rare event, this model is almost free from the programmer's point of view.

#### *5.2.2. Coarse grained locking*

Programmers find this model somewhat difficult, but doable after a short ramp up period. The classes of locks are small and cover broad areas of the kernel; adding a lock to the kernel is a rare event, so the task of the programmer is typically limited to making sure that there is a lock somewhere which is covering the data at hand. So long as the number of locks (or lock classes such as a file lock, or socket lock) is small, the time it takes to find the right lock is also small.

The work that a programmer does in this model is mostly "up front thinking." In other words, the programmer must understand the dozen or so different lock classes in the system and the parts of the system covered by each. Locks are not added to the system as a normal part of enhancement or maintenance, locks are added only when a new "object class," such as a file or a socket, is added to the system. Since adding new classes is a rare event, adding new locks is also a rare event. The programmer needs to learn which lock it is that protects the area in which she is working, and make sure that lock is taken.

This is essentially a slight refinement of the single lock model. In a coarse grained locking model, we have a single lock per class or region in/of the kernel. New locking calls are added only when new entry points to that class/region are added.

#### *5.2.3. Fine grained locking*

A fine grained system will have a large number of locks: Solaris and IRIX are good examples of such systems. In these systems, locking can be (and frequently is) at the individual data structure level, i.e., each element of a list.

Programmers usually think that this is going to be easy, only to find out it is quite hard. The problem is that as the number of locks goes up, the chances for deadlocking also goes up (i.e., there is a chain of locks to get to point A, then the code wants to go to point B but one of the locks on the way to point A is also on the way to point B, so the process will hang forever waiting for a lock it already has; this is actually not that hard to work around, but the two process case of this can be impossible to work around).

Working in such a system requires substantially more knowledge and care on the part of the programmer. The programmer has to avoid deadlocks - a non trivial task since the number of locks can approach 10,000. The programmer also has to maintain scaling, i.e., adding an important code path that is only slightly scalable is a no-no in a fine grained system.

### *5.3. Programmer locking tendencies*

The real question, with respect to locking, is what do programmers tend to do in each of the models? The answer to this question will lead us to the so called locking cliff. This section examines what an average programmer will do in most of locking models.

#### 5.3.1. Single lock

The natural tendency will to be to do no locking, it's been done already. As previously mentioned, all they need to do is to make sure they don't add an entry point without taking the lock as part of the entry point.

#### 5.3.2. Coarse grained locking

The natural tendency will to be to do no locking, it's been done already. New locks are added only for new objects.

#### 5.3.3. Fine grained locking

The programmer's natural tendency in this system will be to add a lock at every step. It is far easier to add the lock and hope that it doesn't create a dead lock than to figure out if a lock already exists and covers this data structure at a higher level, which leads us over the cliff.

## 6. The locking cliff

The locking cliff is the point at which it becomes easier to add another lock to the system instead of determining which existing lock is already protecting the data structures in question. Once the cliff is reached, average (and even above average) programmers will add locks every time any sort of data structure is added to the system.

Once this cliff is reached, the number of locks in the kernel will increase at a dramatic rate, virtually guaranteeing no return from the cliff. In fact, we know of no OS which has ever reached the cliff and then moved back from the cliff.

Since locks are not free, an exponential increase in the number locks will have a noticeable affect on performance and complexity. It is this affect which is the basis for the claim that SMP scaling can be considered harmful. The next section tries to quantify that harm.

## 7. Cost vs. benefit of scaling

Since the benefits of scaling are widely known, we only briefly touch on them here. The costs of scaling are nowhere near as widely known so they are discussed in more detail.

### 7.1. Benefits

The benefit of scaling has always been clear: larger problems can be solved more quickly on one machine. The classes of problems which require the OS to scale include databases, web serving, software development (make), and scientific computing - and this list is by no means complete.

In the competitive world of computing, it is a substantial marketing advantage to be able to say that your OS scales on a 256 processor system and the competition's does not.

### 7.2. Costs

#### 7.2.1. Performance cost

It seems strange to say that there is a performance cost associated with scaling up, but there is. The scaling can (and does) negatively impact the performance of the smaller machines. Consider a 2 processor system.[3] In order to scale up a 2 processor system, only a few locks are needed to achieve perfect scaling. The same OS needs dramatically more locks to scale to 256 processors. Since those locks are unnecessary for the smaller machines, the smaller machines are penalized for the benefit of the larger machines. Given that one and two processor machines make up over 99% of the market, the OS is penalizing the common case for the benefit of the extremely uncommon case.

The effects of scaling on smaller machines are measurable, but it is not easy to quantify these effects - in order to do that correctly, we would need the same OS with and without the locks necessary for scaling up to many processors. The closest we have to that is Linux running on MIPS or SPARC hardware and being compared to IRIX or Solaris on the same hardware. Micro benchmarks, such as lmbench, have shown the commercial operating systems to be substantially slower than Linux, but that is only somewhat interesting - micro benchmarks are not always the best measure of application performance. In one real test, running the BitKeeper regression test[4] one of the above operating systems was shown to be 5 times slower than Linux when running the same test.

### 7.2.2. Software cost

There is widespread agreement in the industry that multi threading an operating system is a difficult task, and that it gets more difficult as time goes on (and the number of locks go up). Getting Solaris to work reasonably well as a multi threaded OS, for example, took about 6 years.

The locking related complexity leads to longer development schedules. We can't quantify this - there is no data for the same OS with and without locking. However, we can look at how long it takes to fix the same sorts of problems in Linux vs. the commercial operating systems. In both, trivial bugs can be fixed quickly;[5] complicated bugs, however, tend to take substantially longer to fix in the commercial operating systems. This author, as a kernel developer in both Solaris and IRIX, can provide personal support for the idea that locking adds complexity and development time.

### 7.2.3. Hardware cost

One frequently overlooked area is the effect of scaling on the hardware. Scaling is not free - scaling requires more locks and those locks generate coherency traffic through the coherency fabric (which could be a snoopy bus or something else, such as SGI's directory based NUMA design). When the locking traffic goes up, the user application has to share the coherency fabric with the kernel's lock traffic, resulting in less useful work done for the user.

A crucial point to realize is that most of the lock traffic is unnecessary. Think back to the run queue example: the goal there was to partition the problem such that there was never a lock which could not be taken - in other words, the problem was split up into N smaller problems with little or no communication amongst the sub sections. If the partitioning is perfect, then each sub section has a CPU and that CPU's locks are only interesting to itself - no other CPU will

[3] Or a uniprocessor system if the MP OS is run on the uniprocessor.

[4] http://www.bitkeeper.com - the test is a large shell script which creates, modifies, deletes many small files.

[5] SGI is known for being able to go from a bug report to patch in less than a day.

 

ever attempt to take those locks. Unfortunately, the locks participate in the coherency protocols whether the OS likes that or not - there currently is no way to say “this memory is specific to this CPU only[6]”

The point of this section is to make people realize that the coherency traffic is required but much of it is unnecessary due to partitioning. The hardware people talk about “false sharing” to describe a similar concept, I suppose this could be called “false coherency.”

### *7.3. Cost / benefit comparison*

The first thing to realize is that regardless of the exact cost of scaling, it benefits a tiny percentage of the total user base. If we could say that scaling is 90% of the work and that large scale MP hardware represents 1% of the user base, then that means 90% of the work is getting done for 1% of the users of the operating system. That might be acceptable if the work had no negative impact on the other 99% of the user base. There is a measurable effect on the rest of the users; the locks hurt performance as well as time to market (due to complexity issues).

In an ideal world, the amount of work done would be proportional to the number of people benefiting from that work.

[6] But one hardware vendor has hinted strongly that they are building hardware with at least two levels of coherency to solve this problem.

## 8. What should we do?

We can’t abandon the big systems, they are far too important, and those 1% customers have a lot of money. It is not acceptable to say “just stop doing any more threading when you get to 4 processors.”

Scaling via the conventional multi threaded OS is not the only way to use big hardware. Another way might be to scale the OS up to a small number of processors, and then cluster multiple copies of that OS on one machine. The OS would have to be extended to work well on a one machine “cluster.” The details of how to do that are beyond the scope of this paper, but are available as a clustering roadmap found in a companion paper.

 