Considerations in software design for multi-core multiprocessor architectures

Today's chip multithreaded, multi-core, multiprocessor systems provide software designers a great opportunity to achieve faster and higher throughput.However,there are a few key design considerations, if ignored, could result in hard-to-find performance issues and scalability bottlenecks.These key design considerations are discussed in this article.

Share:

Gurudutt Kumar is a software developer in the IBM Rational ClearCase - Multi-Version File System team and has nine years of industry experience in the field of operating systems, file systems and kernel programming.

20 May 2013

Also available in Chinese Russian

Introduction

Computing hardware is fast evolving. Transistor density is increasing while clock speeds are leveling off. Processor manufactures are looking to increase the multiprocessing capabilities by having more cores and hardware threads per chip. For example, IBM POWER7® symmetric multiprocessor architecture achieves massive parallelism by supporting up to 4 threads per core, 8 cores per chip and 32 sockets per server. That is a total of 1024 simultaneous hardware threads. In comparison IBM POWER6® architecture supported only 2 threads per core, 2 cores per chip and 32 sockets per server, a total of 128 parallel hardware threads.

While developing software, designers are now required to consider the multiprocessor, multi-core architectures that the software might be deployed on. This is because:

- Applications are expected to perform better and scale better by using more cores, hardware threads, higher memory and meet the growing demands for performance and efficiency.
- With the increased use of multi-core, multiprocessor systems, software design considerations are now expected to include methods to efficiently distribute the software functionality across these computing resources.
- Applications meant to be run in multiprocessor, multi-core environments could end up with serious, hard-to-find performance issues if these considerations are not addressed during design.

This article outlines some of the key considerations for designing software for multi-core, multiprocessor environments.

Impediments to software scalability on chip multithreaded, multi-core, multiprocessor architectures

Applications are expected to scale and perform better on multi-core, multiprocessor environments. However, an inefficiently designed application might perform poorly on such an environment rather than scaling well and performing better by using the available computing resources. Some of the key impediments to this scalability could be:

- **Inefficient parallelization:** A monolithic application or software will not be able to use the available computing resources effectively. You need to organize the application into parallel tasks. This issue is regularly seen in legacy applications or software that do not support multithreading. These applications

fail to scale and achieve better throughput on multi-core, multi-processor, chip multi-threaded hardware. Too many threads could also be as bad as too few threads.

Serial bottlenecks: Applications that share data structures among multiple threads or processes could have serial bottlenecks. In order to maintain data integrity, access to these shared data structures might have to be serialized by using the locking and serialization techniques, for example, read lock, read-write lock, write lock, spinlock, mutex, and so on. Inefficiently designed locks could create serial bottlenecks due to high lock contentions between multiple threads or processes trying to acquire the lock. This could potentially degrade the performance of the application or software. The performance of an application could degrade as the number of cores or processors increase.

Overdependence on operating system (OS) or runtime environments: You cannot depend on the OS, runtime environments or compilers to do everything needed to scale the application or software. Although, compilers and runtime environments could aid in optimizing to a certain level, you cannot depend on it to address all the scalability issues. For example, cannot depend on Java™ Virtual Machine (JVM) to discover opportunities for better scalability of a Java application by parallelizing automatically.

Workload imbalance could be a bottleneck: An unevenly distributed workload can be inefficient in utilizing computing resources. You might have to break large tasks into smaller ones that can be run in parallel. You might have to change serial algorithms into parallel ones for improving performance and scalability.

I/O bottlenecks: Bottlenecks due to blocking disk input/output (I/O) or high network latencies could severely inhibit application scalability.

Inefficient memory management: On multi-core platforms, pure computing might be cheap because there are many processing units and primary memory might also not be a problem because it is getting

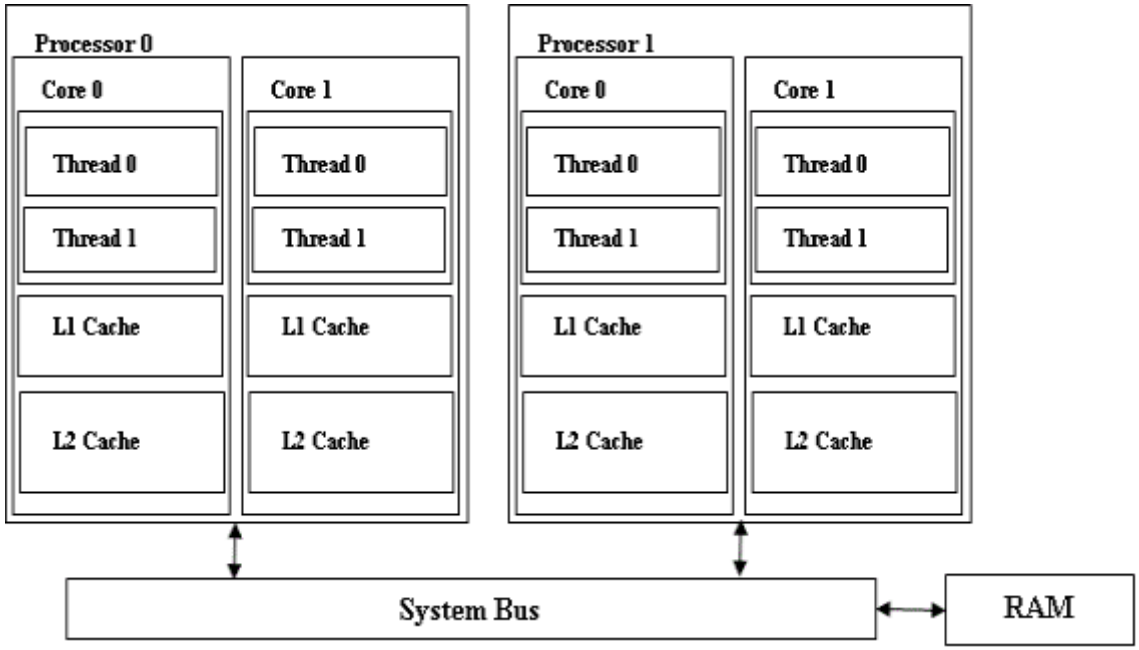
larger. However, memory bandwidth remains the bottleneck all the time because all processor cores share a common bus. Inefficient memory management could cause hard-to-detect performance issues such as false sharing.

Low processor utilization could obviously mean suboptimal resource utilization. In order to understand performance issues, you need to evaluate if an application has too few or too many threads, locking or synchronization issues, network or I/O latencies, memory thrashing or other memory management. High processor utilization is normally good as long as resources are spent in application threads doing meaningful work.

Overview of chip multithreaded (CMT), multi-core, multiprocessor (MP) systems

Before discussing the design considerations for chip multithreaded, multi-core, multiprocessor environments lets take a brief look at such a system. The system depicted in Figure 1 has two processors, each with two cores and each core has two hardware threads. There is one L1 cache and one L2 cache per core. As such, each core could have its own L2 cache or the cores on the same processor could share the L2 cache. Hardware threads on the same core share L1 and L2 cache.

Figure 1. A typical chip multithreaded, multi-core, multiprocessor system



All the cores and processors share the system bus and access the main memory or RAM through the system bus. For applications and the operating system, this system looks like eight logical processors.

The following key concepts will help us understand the challenges in designing applications for such a chip multithreaded, multi-core, multiprocessor environment.

Cache coherency

Cache coherency is a state where the value of a data item in a processor's cache is the same as that in system memory. This state is transparent to the software. However, the operations performed by the system to maintain cache coherency can affect the software's performance.

Consider the following example. Let's assume that Thread 1 is running on processor 0 while Thread 2 is running on processor 1 of the system depicted in Figure 1. If both these threads are reading and writing to the same data item, then the system has to perform extra operations to ensure that the threads are seeing the same data value as each read and write operation occurs.

When Thread 1 writes to the data item that is shared with thread 2, the data item is updated in its processors cache and in system memory but not immediately updated in the cache of thread 2s processor because thread 2 might no longer require access to the data item. If thread 2 then accesses the data item, the cache subsystem on its processor must first obtain the new data value from system memory. So, the write by thread 1 forces thread 2 to wait for a read from system memory the next time that it accesses the data. This sequence only takes place if the data is modified by one of the threads. If a series of writes is done by each thread, the performance of the system can seriously degrade because of all of the time spent waiting to update the data value from system memory. This situation is referred to as *"ping ponging"* and avoiding it is an important software design consideration when running on multiprocessor and multi-core systems.

Snooping

It is the cache subsystem that keeps track of the state of each of its cache lines. It uses a technique called *"bus snooping"* or otherwise called *"bus sniffing"* to monitor all transactions that take place on the system bus to detect when a read or write operation takes place on an address that is in its cache.

When the cache subsystem detects a read on the system bus to a memory region loaded in its cache it changes the state of the cache line to **"shared"**. If it detects a write to that address, then the state of the cache line is changed to **"invalid"**.

The cache subsystem knows if it has the only copy of the data in its cache as it is snooping the system bus. If the data is updated by its own CPU, the cache subsystem will change the state of the cache line from **"exclusive"** to **"modified"**. If the cache subsystem detects a read to that address from another processor, it can stop the access from occurring, update the data in system memory, and then allow the processor's access to continue. It will also mark the state of the cache line as **shared**.

Refer to the article on "Software Design Issues for multi-core multi-processor systems" in the Resources section for more information on these concepts.

Influence of multi-core, multiprocessor environments on software design decisions

When designing software to run on a multi-core or multiprocessor system, the main consideration is how to allocate the work that will be done on the available processors. The most common way to allocate this work is by using a threading model where the work can be broken down to separate execution units that can run on different processors in parallel. If the threads are completely independent of one another, their design does not have to consider how they will interact. For example, two programs running on a system as separate processes each on its own core do not have any awareness of each other. Performance of the programs is not affected unless they contend for a shared resource such as system memory or the same I/O device.

Main focus of the discussion that follows will be on the way that the cores and processors interface with the main memory and how this has an impact the software design decisions.

See the following key design considerations.

Avoid memory contention

Different cores share a common data region, in memory and cache, which needs to be synchronized among them. *Memory contention* is when different cores concurrently access the same data region. Synchronizing data among different cores has big performance penalty because of bus traffic, locking cost and cache misses.

If an application has multiple threads and all the threads are updating or modifying the same memory address, then, as discussed in the previous section, there could be significant ping-ponging in order to maintain cache coherency. This will lead to degraded performance.

For more details, refer to the "Memory Contention" section in the article "Memory issues on multi-core platforms" in the Resources section. The article includes a simple program that demonstrates the ill effects of memory contention. The example demonstrates that even if only one variable is shared among multiple threads, the performance penalty could be very significant even when atomic instructions are used for updates.

Techniques to avoid memory contention

Don't share writable state among cores:

To minimize memory bus traffic, core interactions need to be kept as low as possible by minimizing shared locations/data, even if the shared data is not protected by lock but some hardware level atomic instructions such as InterlockedExchangeAdd64 on the Microsoft® Windows® 32-bit platform.

One way to reduce memory contentions between threads is to eliminate updates to a shared memory region from multiple threads. For example, even when a global counter or cumulative total, such as a statistic, is required to be updated by multiple threads, the individual threads could maintain thread local totals and have the global total updated by a common thread only when it is required. Thereby, the contention on the shared memory region is reduced significantly.

The patterns that tend to reduce lock contention also tend to reduce memory traffic, because it is the shared writable state that requires locks and generates contention.

Avoid false sharing caused by core cache. Refer to the next section for details.

Avoid false sharing

If two or more processors are writing data to different portions of the same cache line, then a lot of cache and bus traffic might result for effectively invalidating or updating every cached copy of the old line on other processors. This is called *"false sharing"* or also *"CPU cache line interference"*. Unlike true sharing where two or more threads share the same data (thereby needing programmatic synchronization mechanisms to ensure ordered access), false sharing occurs when two are more threads access unrelated data which resides on the same cache line.

Consider the following code snippet to understand false sharing better (Ref: False Sharing).

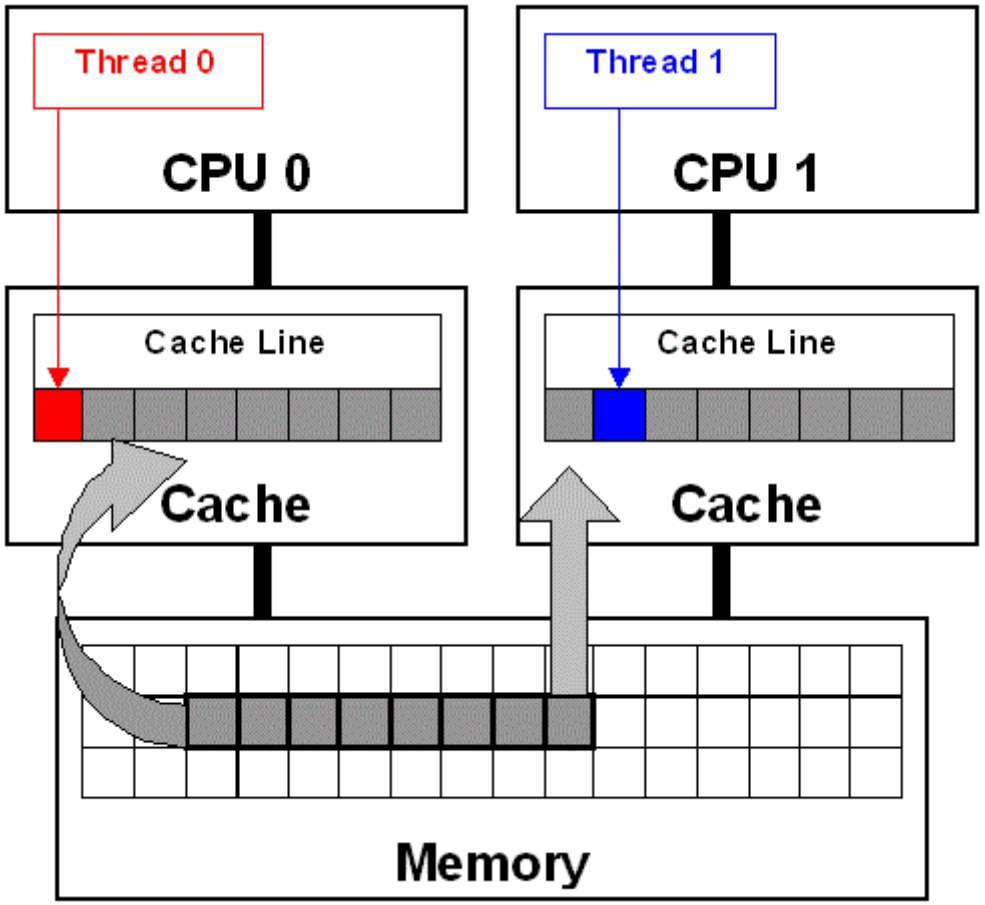
Listing 1. Sample code to illustrate false sharing

```
double sumLocal[N_THREADS];
. . . . .
. . . . .
void ThreadFunc(void *data)
{
      . . . . . .
      int id = p->threadId;
      sumLocal[id] = 0.0;
      . . . . . .
      . . . . . .
      for (i=0; i<N; i++)
      sumLocal[id] += p[i];
      . . . . . .
}
```

In the above code example, the variable, sumLocal is of the same size as the number of threads. The array sumLocal can potentially cause false sharing as multiple threads write to the array when the elements that they are modifying lie on the same cache line. Figure 2 illustrates the false sharing between thread 0 and thread 1 while modifying two consecutive elements in sumLocal. Thread 0 and thread 1 are modifying different but consecutive elements in the array sumLocal. These elements are adjacent to each other in memory and hence will be on the same cache line. As illustrated, the cache line is loaded into caches of CPU0 and CPU1 (grey arrows). Even though the threads are modifying different

areas in memory (red and blue arrows), in order to maintain cache coherency the cache line is invalidated on all the processors that have loaded it, forcing an update.

Figure 2. False sharing (Ref: Avoiding and identifying false sharing among thread inResourcessection.)



False sharing could severely degrade performance of the application and it is hard to detect. Refer to the article "Memory issues on Multi-core platform - CS Liu" in the Resources section for a simple program that demonstrates the ill effects of false sharing.

Techniques to avoid false sharing

False sharing could be avoided by aligning data structures to cache line boundaries using compiler alignment directives available for conditionally compiling for a particular processor. For example, on he Linux® platform, the header file adm-i386/cache.h defines the macro, L1_CACHE_BYTES, for L1 cache line size for the Intel® x86 architecture family. The processor cache line size could also be determined programmatically. Refer to the Resources section for details on aligning data structures on cache line boudaries and for a cross-platform function to get the cache line size programatically.

One other technique involves grouping the frequently accessed fields of a data structure so that they fall into a single cache line and thereby they can be loaded with a single memory access. This can reduce memory latency. However, this can increase the cache footprint if the data structures are huge and might have to compromise on some packing efficiency to reduce or eliminate false sharing. Refer to the article "Elements of Cache Programming Style" in the Resources section for details.

In order to prevent false sharing in arrays, the base of an array should be cache aligned. The size of a structure must be either an integer multiple or an integer divisor of the processor cache line size.

If it is needed to assume a cache line size for enforcing alignment, then use 32 bytes. Note that:

32 byte aligned cache lines are also 16 byte aligned.

On vast majority of processors assuming 32 byte cache line size would be appropriate. For example, IBM PowerPC® 601 nominally has a 64 byte cache line but it really has two connected 32 byte cache lines. The Sparc64 has a 32 byte L1 and a 64 byte L2 cache line. Alpha has a 32 byte L1 cache line. Itanium architecture has 64 byte L1 cache lines, IBM System z® has a 256K L1 cache and 128 byte cache lines and x86 processors have 64 byte L1 cache lines.

The general rule for efficient execution on a single core is to pack data tightly, so that it has a small a footprint. But on a multi-core processor, packing shared data can lead to a severe penalty from false

sharing. Generally, the solution is to pack data tightly, give each thread its own private copy to work on, and merge results afterwards.

Pad structures or data used by threads to ensure that data owned or modified by different threads lie on different cache lines.

False sharing is hard to detect but there are a few tools such as Oprofile and the Data Race Detection (DRD) module of Valgrind that could be of help.

Eliminate false sharing? Wrong!
The objective should ideally be to eliminate sharing not just false sharing. Software design should try to eliminate the need for locks and synchronization mechanisms and sharing in general. See this article by Dmitriy Vyukov for the important perspective.

Eliminate or reduce lock contentions

This design consideration is an extension of the previous two considerations to avoid memory contentions and false sharing. As discussed in the previous section, the primary goal of the software designer should be to eliminate sharing so that there is no contention for resources between threads or processes. Some of the techniques discussed in the previous sections such as use of thread local variables instead of global shared regions could prevent both memory contentions and false sharing. However, this technique might not be applicable in all cases.

For example, if there is a data structure that maintains the state of a resource, then it might not be possible to have copies of it in each thread. This data structure might have to be read and modified by all threads in the application. Hence, synchronization techniques would be necessary to maintain data coherency and integrity of such shared data structures. When ever there are locks or synchronization

constructs protecting a shared resource, then there might be contentions for the locks between multiple threads or processes potentially degrading performance.

On a multi-core, multiprocessor system, there might be room to run a significantly large number of threads or processes concurrently, however if these threads have to constantly contend with each other to access or modify shared resources or data structures then the overall throughput of the system drops. This results in the application not being able to scale to utilize the available computing resources efficiently. In the worst cases of performance degradation due to lock contentions, as the number of cores or processors increase, the performance of an application could drop.

Techniques to avoid lock contention

One of the methods to avoid lock contentions in data structures is to adopt concurrent data structure designs and lock free algorithms that eliminate locks and traditional synchronization techniques like mutex. There are several designs of concurrent data structures available that do not need to employ synchronisation mechanisms such as mutex. Refer to the Resources section for few examples of such concurrent data structures designs.

Some examples of lockless algorithms are:

Scalable concurrent hash tables using Relativistic Programming: The earliest example of this technique is Read Copy Update (RCU), which is used extensively in the Linux kernel. It has led to substantial performance improvements and code simplifications in Linux kernel.

Lock-free extensible Split-ordered Hash lists: This lock-free recursive extensible hashing algorithm uses lock-free linked lists that use atomic instructions for modifications of the linked list.

In the Linux kernel, there is extensive use of per-processor variables where each processor on the system gets its own copy of the given variable. Access to per-processor variables does not need locking

and as these variables are not shared among threads on different processors, there is no false sharing or memory contention. This technique is ideal for statistics collection.

Techniques to reduce lock contention

When traditional locking or synchronization techniques such as spinlocks are used, care has to be taken to not have monolithic or global locks, instead have to break them into granular pieces. Thereby the locks protect a specific and small area of the data structure. This enables multiple threads to concurrently operate on different members of the same data structure by acquiring the corresponding lock protecting those members. This way more concurrency can be achieved.

Even when the synchronization mechanism in the software design achieves better concurrency and reduced lock contention, there could be performance issues due to false sharing. For example, consider a hash data structure. If there is an array of spinlocks, protecting each of the buckets in the hash then, there could be false sharing in the array of spinlocks. Two threads, running on two different processors, each locking a different bucket in the hash, could have false sharing if the spinlocks that they acquire happen to reside on the same cache line. Hence the general techniques to avoid false sharing need to be considered in the design of such algorithms.

Detecting lock contentions and eliminating or reducing lock contentions is important for improving scalability of applications on multi-core, multiprocessor environments. Operating systems provide utilities to detect and measure performance bottlenecks due to lock contentions. For example, Solaris provides the Lockstat utility to measure the lock contentions in kernel modules. Similarly the Linux kernel provides the Lockstat and Lockdep framework for detecting and measuring lock contentions and performance bottlenecks. **"Windows Performance Toolkit - Xperf"** provides similar capabilities on Windows. See Resources section for more details.

Avoid Heap Contention

C/C++ standard memory management routines are implemented using platform-specific memory management APIs, usually based on the concept of heap. These library routines (whether it is single thread version or multi-thread version) allocate or free memory on a single heap. Its a global resource that is shared and contended among threads within a process. This heap contention is one of the bottlenecks of multi-threaded applications that are memory intensive.

Technique to avoid heap contention

> To use thread local/private heap to do memory management, thus the resource contention is eliminated. On Windows platform, a dedicated heap for each thread can be created using HeapCreate() and pass the returned heap handle to HeapAlloc()/HeapFree() functions.

The article "Memory issues on Multi-core platform - CS Liu" in the Resources section provides an example on heap contention. In the reference, the author demonstrates that using private heap will result in around 3 times performance gain as compared to using the global heap.

Note:

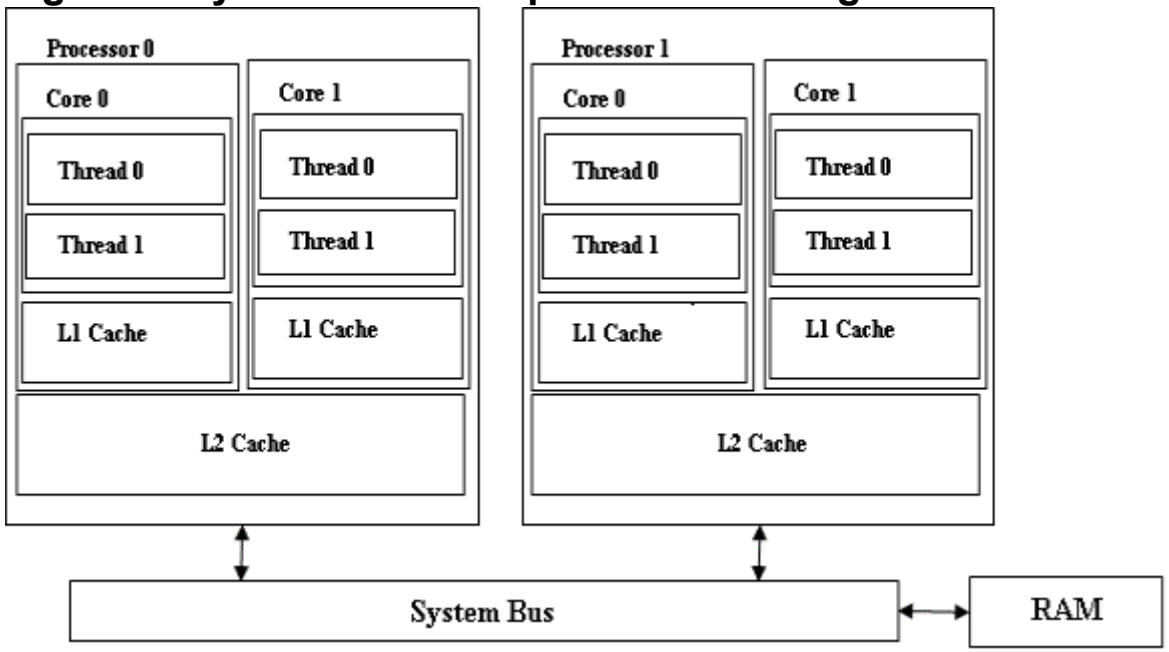
> On Windows platform, the heap_no_serialization flag can be set when creating a heap, this means that there will be no synchronization cost when accessing it from multiple threads. But it turns out that setting this flag to thread private heap will be very slow on vista and later operating system.
>
> Heap_no_serialization and some debug scenarios will disable the Low Fragment Heap feature, which is now the de facto default policy for heaps and thus highly optimized.

Improve Processor Affinity

Processor affinity is a thread or process attribute that tells the operating system which cores, or logical processors, a process can run on. This is more applicable in embedded software design.

Figure 3 shows a system configuration with two processors each with two cores that have a shared L2 cache. In this configuration, the behavior of the cache subsystems will vary between two cores on the same processor and two cores on different processors. If two related processes or threads of a process are assigned to the two cores on the same processor then they can take advantage of the shared L2 cache better and lower overhead of maintaining cache coherency.

Figure 3. System with multiple cores sharing an L2 cache



Design considerations

Designers of software for embedded systems could take advantage of this relatively low overhead of maintaining cache coherency, by programmatically controlling the allocation of threads to cores.

The following system calls on Linux and Windows operating systems can tell the application what the processor affinity is for a particular process and can set the processor affinity mask for a process:

Listing 2. System calls for managing processor affinity

```
Linux Example:
/* Get a process' CPU affinity mask */
extern int
sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *cpuset);

/* Set a process's affinity mask */
extern int
sched_setaffinity(pid_t pid, size_t cpusetsize, const cpu_set_t *cpuset);

Windows Example:
/* Set processor affinity */
BOOL WINAPI
SetProcessAffinityMask(Handle hProcess, DWORD_PTR dwProcessAffinityMask);

/* Set Thread affinity */
DWORD_PTR WINAPI
SetThreadAffinityMask(Handle hThread, DWORD_PTR dwThreadAffinityMask);
```

Programming models

A software designer can consider two different programming models when assigning work to threads in an application. They are:

Functional decomposition

The objective of this model is to understand the operations that the application or software has to perform and assign each operation to a distinct thread.

Multiple operations can be grouped in a thread but the result of functional decomposition is that each operation is performed by a particular thread.

Domain decomposition or otherwise called *data decomposition*

The objective of this model is to analyze the data sets that the software or application needs to manage or process and to break those into smaller components that can be handled independently.

Threads in the software replicate the operations to be performed but each thread will operate on a separate data component.

The types of operations and characteristics of the data that the software is required to operate on, influences the choice of the model, but it is required to understand how these models perform in multiprocessor or multi-core environments.

Points needed to be considered by software designers

Although the apparent simplicity of cache interactions in data decomposition might suggest that it might be preferred over functional decomposition, software design challenges required to support domain decomposition can be significant.

Data in the decomposition model still needs to be on different cache lines to avoid the potential of false sharing.

The reference **Software Design Issues for Multi-core Multi-processor Systems** in the Resources section has detailed description of the programming models, challenges in applying them in software design for multi-core, multi-processor architectures and their advantages.

Conclusion

The article briefly explained the characteristics of multi-core, multithreaded environments. Some of the key issues that could cause performance degradation or impede scalability of applications on multi-core, multiprocessor systems were also touched upon. Some of the key considerations and techniques in designing software for such environments were also discussed. Software or application designed with these considerations could utilize the available computing resources efficiently and also avoid performance and scalability issues.

Resources

- Software Design Issues for Multi-core Multi-processor Systems
- Memory issues on multi-core platform - CS Liu
- Avoiding and Identifying False Sharing Among Threads
- Align Data Structures on Cache Boundaries
- Cross platform function to get cache line size (Author: Nick Strupat)
- Elements of Cache Programming Style - Chris B Sears
- Designing concurrent data structures without mutexes – Arpan Sen (developerWorks)
- lockstat - Measuring lock spin utilisation on Solaris
- Lock Statistics Documentation - Linux Kernel
- Windows Performance Toolkit - Xperf

Dig deeper into AIX and Unix on developerWorks

- Overview
- Technical library (tutorials and more)
- Forums
- Community
- Downloads and products
- Open source projects
- Events

developerWorks Premium
Exclusive tools to build your next great app. Learn more.

dW Answers
Ask a technical question

Explore more technical topics
Tutorials & training to grow your development skills