Contents

3.1 Speculative State

The memory wall is an obstacle to obtaining scalable speed-up from the execution of a program on a Chip Multi-Processor. Transactional Memory systems promise to speed-up concurrent execution by removing the barriers to scalability imposed by mutual exclusion, but concurrent speed-up has only been demonstrated in a few applications, because the buffering of speculative state increases the memory bandwidth requirement of a concurrent program, restricting scalability. The use of immutable memory permits concurrent programs to scale to greater numbers of processors before hitting the memory wall.

The effective memory bandwidth of a scalable concurrent program must be independent of the number of processors participating in its execution.

The main contribution of this section is an examination of why the demonstrable concurrent speed-up of general applications has remained elusive. This section focuses on identifying the impact of the buffering of speculative state on memory bandwidth as a factor limiting the speed-up that can be achieved.

3.1.1 The Memory Wall

For execution-bound programs there is a potential for speed-up from concurrent execution on a Chip Multi-Processor. For such programs a barrier to concurrent speed-up is mutual exclusion as described by Amdahl's law. Speculative execution avoids the need for mutual exclusion and alleviates the scaling restrictions of Amdahl's law. However, the scaling of a concurrent program is bounded by restrictions imposed by both memory latency and memory bandwidth. Wulf describes these restrictions which are collectively known as the memory wall [WM95].

The connection between the processors of a Chip Multi-Processor and main memory has a finite bandwidth that is shared by all of the processors. The connection consists of the caches, the memory controller and the wiring between the processor chip and main memory. Contention in the common components of the path to memory affects the speed at which memory requests can be serviced.

A program has a memory bandwidth requirement which is the bandwidth, expressed in bytes per second, that it consumes. An increase in the memory bandwidth requirement leads to an increase in the latency of individual memory requests and an increase in the elapsed execution time of the program [HP06].

In a Chip Multi-Processor a finite memory bandwidth is shared amongst all of

the processors and this limits the speed-up that can be obtained from concurrent execution. Increasing the available memory bandwidth is a much more difficult engineering challenge than increasing the number of processors in a Chip Multi-Processor so memory bandwidth tends to increase more slowly than aggregate processing power.

Section 3.1.3 describes the limiting effect of memory bandwidth on execution time and the difficulty of increasing the available bandwidth.

Buffering speculative state increases the memory bandwidth of a concurrent program. Data is written twice, once as isolated speculative values and again as shared committed values. Bookkeeping information, required to ensure correct concurrent execution, is also written to memory. Wasted work from failed speculation also contributes to the volume of data written to memory. Together these factors cause a concurrent program to have a much higher memory bandwidth requirement than the equivalent serial program.

Section 3.1.4 describes how storing speculative state increases the memory bandwidth requirement of a program.

For many applications the memory wall is a constraint on the speed of serial execution and such applications are known as memory-bound. It is reasonable to expect that memory bandwidth will also be the main barrier to obtaining concurrent speed-up on Chip Multi-Processors. The use of multiple processors does little to alleviate the memory wall problem, instead Chip Multi-Processors make the memory bandwidth problem more acute.

Section 3.1.5 describes how concurrent programming transforms an execution-bound program into a program bounded by memory bandwidth.

Chip Multi-Processors enforce a cache coherency protocol to keep caches coherent but mechanisms to ensure cache coherency do not scale well. The overheads associated with maintaining coherent caches reduce the effectiveness of caching and thus increase the effective memory bandwidth of a concurrent application. The engineering difficulty of scaling cache coherency mechanisms is a barrier to increasing the number of processors in a Chip Multi-Processor design.

Section 3.1.6 describes the difficulty of scaling the mechanisms that ensure cache coherence.

3.1.2 Immutable Memory

When only immutable data is used to represent shared state, the amount of shared data that is either read or written to main memory by a program is independent of the number of processors involved in its concurrent execution.

A concurrent program should maintain both speculative and shared state immutably in memory. Immutable values are written just once so immutable data satisfies the requirement that it does not increase memory bandwidth of a program. Immutable values cannot change so cached copies are always coherent.

Immutability is a memory usage convention. A memory location is said to be immutable if its value is written just once and cannot be changed thereafter. Prior to writing the value the memory location cannot be reached by the program, so the program cannot read memory locations that have not already been written and cannot write to those locations that have already been written.

An immutable object is an object whose state cannot be modified once it has been created. It can be regarded as a set of constant values. An object reference associates an identifier with a location in memory where the object can be found. An immutable object cannot be modified but a reference to it may be mutable, so an identifier can be associated with different versions of an immutable object by modifying its reference. A concurrent program that maintains state immutably requires mutable memory to maintain both unshared state and shared references to immutable objects.

Immutable objects can be relocated while retaining the property of immutability. To relocate an immutable object in memory a copy of the object is made at another location. Values can be inserted into and deleted from an immutable object during the copy operation. It is possible to create an immutable object with identical properties to any mutable object by implementing all of the object's mutating methods as constructors of new copies of the object. A serial program that maintains state in immutable objects may have a different memory bandwidth requirement from a similar program that uses mutable objects but in many cases immutable objects can be implemented just as efficiently as their mutable counterparts. It is not necessary to perform a full copy of an object every time a mutating method is called to preserve the property of immutability.

Immutable data is written just once so an immutable value written speculatively does not need to be written again when it is shared. A concurrent program that maintains shared state immutably scales without increasing its

memory bandwidth requirement as the total amount of data both written to and read from memory is unaffected by the number of processors participating in its execution.

An immutable object can never go stale in cache because its value cannot be changed so it is not necessary to ensure that the cached copies of an immutable object are coherent. However, a mechanism to enforce cache coherency is required to ensure that all processors observe an up to date copy of the reference to the immutable data.

Immutable data frees Chip Multi-Processors from the scaling restrictions of cache coherency in two ways. Firstly, it is not necessary for the processor design to enforce a cache coherency protocol for all memory locations, allowing the design to be more scalable. Secondly, the cache pathologies of cache coherency misses and false sharing do not occur and this increases the effective memory bandwidth of the cache.

3.1.3 Memory Bandwidth

A program executing in parallel on two processors requires twice the memory bandwidth of an equivalent program executing on one. The bandwidth requirement for processors executing general applications is around 1GB/s per core. Desktop and server Chip Multi-Processors use single or dual DDRx memory systems. The maximum bandwidth of such an arrangement is less than 10GB/s. Jacob offers a reason why four physical core Chip Multi-Processors are common and eight core systems have yet to appear which is that, unless the memory system is upgraded, an eight core system would perform no better than a four core system [Jac09].

A solution to the problem of restricted bandwidth is to increase the memory bandwidth of the processor. Historically, memory bandwidth has increased more slowly than processor frequency for physical reasons, such as the difficulty of scaling the number of off-chip pins. Increasing the number of off-chip pins is challenging because of their energy requirements and because it increases the complexity of printed circuit boards. Currently, processor frequency is static and the number of processors on a chip is increasing. Jacob describes why the number of concurrent memory operations that a processor's memory controllers can support is much harder to scale than the number of processors on the chip [Jac09].

Memory bandwidth can be increased to match the number of cores, but at significant design cost. A Chip Multi-Processor saturates its memory subsystem once the number of cores multiplied by the bandwidth of the program executing on them reaches a maximum sustainable bandwidth. Jacob finds that the 32 core Niagara Chip Multi-Processor has a memory subsystem that saturates at 25GB/s so, the Niagara processor has a memory bandwidth of less than 1GB/s per core [Jac09].

Memory bandwidth is limited by physical factors and dramatic increases in bandwidth are unlikely in the near future. Consequently, proposals to support concurrent programming should focus on decreasing the effective memory bandwidth requirement of programs.

3.1.4 The effect of speculation

Transactional Memory systems take several different approaches to storing speculative state. Each of these approaches has its own relative merits, which are discussed in detail in the main reference book on Transactional Memory [HLR10]. However, each approach involves writing values to more than one location or writing additional meta-data to memory. The additional memory writes tend to increase the memory bandwidth requirement of the program.

Maintaining state in a recovery log is a common technique in Software Transactional Memory systems. Logging state increases memory bandwidth as each shared value must be written to main memory at least twice. Typically, a system will write the old value of a location to a log before storing the new value. For example, the logTM Software Transactional Memory system maintains the committed state of memory locations that have been written speculatively in a log [MBM+06]. This technique is known as eager versioning. The amount of state written to the log is equal to the amount of speculative state written by the program. The latency of a memory write operation can be reduced by caching the log but, eventually, both the old and new values must be written to main memory as a result of the operation thus increasing the memory bandwidth requirement of the program.

Maintaining speculative state in cache is a technique adopted by some Hardware Transactional Memory systems. For example, the Hardware Transactional Memory proposal of Herlihy and Moss maintains speculative state in a dedicated transactional cache [HM93]. Speculative values are eventually written to main

memory in addition to committed values so the caching of speculative state increases the memory bandwidth of a program. When cache contains both the speculative and committed state of an object the number of distinct objects that it can contain is reduced so the caching of speculative state also increases the memory bandwidth of a program by reducing the effectiveness of cache.

Maintaining speculative state in a buffer is a technique adopted by many Hybrid and Software Transactional Memory systems. Buffering shared state increases memory bandwidth because objects must be copied when they are written. Typically, a buffering Transactional Memory system will copy an entire object to a new location when one of its fields is modified speculatively. The operation usually has low latency because it occurs in cache, but the whole of the copied object must eventually be written to main memory as a result of the operation. Object copying increases the memory bandwidth of the program.

Each of these techniques require additional bookkeeping information to ensure the correct concurrent execution of the program. This information will eventually be written to main memory, increasing the effective memory bandwidth of the program.

Speculative execution necessitates that some transactions will be aborted and the work they did will be wasted. Memory operations performed by this wasted work also increases the effective bandwidth of the concurrent program.

Transactional Memory increases the memory bandwidth requirement of the program. In many cases the overhead of buffering speculative state is the main factor limiting the speed-up that can be achieved from the concurrent execution [Olu07].

3.1.5 Moving the bottleneck

The number of processing cores that it is possible to fit into a single Chip Multi-Processor is expected to increase in future. As the number of cores increases so does the potential speed advantage of concurrent programs over their serial counterparts. Concurrent programming is universally accepted to be difficult but at some point the speed advantage of concurrent execution will make the effort of writing concurrent programs worthwhile.

This familiar argument is based on two questionable assumptions. Firstly, that the difficult of writing concurrent programs is a major obstacle to the adoption of concurrent programming. Secondly, that a concurrent program has the potential

to execute faster on a Chip Multi-Processor than the equivalent serial program.

In many application programming environments, such as the computer games industry, there are enormous financial incentives to improve concurrent performance. In such environments no programmer effort is spared in utilising concurrent execution. The difficult of writing concurrent programs can be overcome by applying many programmers to the task and requiring each of them to think very hard. The real problem is that their efforts are so rarely rewarded by improved performance of the program.

The elapsed execution time of a memory-bound program on a Chip Multi-Processor is equal to or greater than the serial execution time, no matter how many processors are applied to the problem. Only execution-bound programs have the potential for a concurrent implementation executing on a Chip Multi-Processor to execute faster than a serial implementation.

For execution-bound programs there is a potential speed-up from concurrent execution. The first obstacle to realising this speed-up is that executing on multiple processors increases the bandwidth of the program causing it to become memory-bound. The second obstacle is that instrumentation to support speculative execution increases the effective memory latency and bandwidth of the program causing it to become memory-bound.

At best Transactional Memory converts a concurrent program with speed-up restricted by mutual exclusion into a concurrent program with speed-up restricted by the memory wall. Transactional Memory systems increase the memory bandwidth of the program and this lowers the amount of scaling possible before a concurrent program hits the memory wall. Programs that have a low memory bandwidth requirement tend to scale well when the restrictions of mutual exclusion are removed and these are the programs that Transactional Memory research focuses on [PW10].

3.1.6 Cache Coherency

Small memories are generally faster than large memories because they contain shorter wires. Processors maintain a hierarchy of caches of different sizes to reduce memory latency and increase memory bandwidth. Chip Multi-Processors maintain both shared and unshared caches. Typically, each processor has a small local cache that is not shared and if a memory access cannot be satisfied from this cache an attempt is made to satisfy it from a larger slower cache shared between

all of the processors of the Chip Multi-Processor.

To present a consistent view of memory to each processor a Chip Multi-Processor implements a cache coherency mechanism which enforces a cache coherency protocol. A snoop-based cache coherency mechanism broadcasts the address of memory locations that have been modified to all caches and a directory-based mechanism records where all of the copies of a particular location reside. Chip Multi-Processors generally enforce snoop-based protocols to avoid the additional latency of accessing a centralised directory.

The implementation complexity of snoop-based cache coherency protocols increases with processor count because the number of processors that can access a memory bus is physically limited, so designers face the challenge of maintaining coherency without the benefit of a single bus to serialise events [Sto06].

A coherency cache miss is a cache miss required to maintain coherency between processor caches. When a cached location is modified by a processor all of the copies of that location held in the local caches of the other processors must either be updated or discarded. Typically, a snoop-based protocol regards the copies held by the other processors as stale and marks them as invalid so the next access to the location will result in a cache miss. Coherency cache misses tend to increase with the processor count and are unaffected by cache size. They have a detrimental effect on performance as each cache miss increases the effective memory bandwidth of the program.

The messages sent between processors to maintain coherent caches are known as coherency bus traffic. Coherency bus traffic increases with processor count and is unaffected by cache size. Congestion on the bus has a detrimental effect on memory latency and additional bus traffic increases the effective memory bandwidth of the program [HP06].

Bibliography

[HLR10] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Jac09] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[Olu07] Kunle Olukotun. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan and Claypool Publishers, 1st edition, 2007.

[PW10] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *ISPASS IEEE International Symposium on Performance Analysis of Systems and Software*, pages 97–108, 2010.

[Sto06] Jon Stokes. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press, San Francisco, CA, USA, 2006.

[WM95] Wm. A. Wulf and Sally A. Mckee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.