Secondary Bus Performance in Reducing Cache Writeback Latency

by

Rakshith Thambehalli Venkatesh

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 9, 2011

Keywords: Cache Writeback, System Bus, Queuing Delay, Processor Performance

Abstract

For single as well as multi core designs, effective strategies to minimize cache access latencies have been proposed by a number of researchers over the last decade. Such designs include the Miss Status Holding Registers, Victim Buffers, Eager and Lazy Write backs, and Cache Pre-fetching. However, write-buffer stalls remain a bottleneck in real-time memory accesses. To alleviate this problem, the Secondary Bus Architecture was developed at Auburn. The secondary bus connects the write back buffer to the main memory via an independent secondary bus controller to retire dirty cache lines to memory. The write back traffic is only about 25-30% of the total traffic between the last level of cache and memory and is intermittent compared to read requests. Therefore, a narrow 8-bit secondary bus was used in the implementation. The secondary bus controller identifies idle main bus cycles by snooping on the main bus control lines. These idle cycles are used to retire write back buffer entries to the main memory.

In this research, we evaluated the effectiveness of secondary bus in retiring cache write-backs to the memory using a series of extensive rigorous experiments run on the computers of the Alabama Super Computer Center using SimAlpha and SPEC CPU 2006 benchmarks. The simulator SimAlpha was used for analyzing the architecture since it incorporates a well defined memory hierarchy. The SPEC CPU 2006 programs are both CPU and memory intensive and thus were ideal candidates for our evaluations. The I/O injections used normal traffic distribution using DMA as well as the new Direct Cache Injection mechanism.

We observed performance improvements of up to 35% over the base architecture (i.e. one without a secondary bus) in presence of I/O traffic on the main bus and 17% in absence of any I/O traffic. Furthermore, queuing delays on the main bus were observed to drastically reduce. In comparisons with

Eager Write back, a strategy that is popular in many contemporary cache designs, it was found that the secondary bus architecture is much superior in performance.

Acknowledgements

My years at Auburn University as a graduate student have been excellent mainly because of a great academic course structure and the emphasis on research. This setup wouldn't have been possible without the excellent faculty and amazing support staff at Auburn University who encourage innovation and provide exceptional classroom, library, laboratory and also recreational facilities. I would like to thank Auburn's faculty and staff members who have helped me amass great amount of knowledge, hone my skills and graduate with a Master's degree.

I would like to thank Professor Sanjeev Baskiyar for supervising my thesis. I also thank him for providing funding for the master's thesis research for two years with a DARPA/AFRL grant. His availability for discussion and advice regarding my research work as well as my career choice is highly appreciated. I would also like to thank him for allowing me to go on an internship for the summer of 2010 and get valuable experience. I appreciate John O'Farrell's help during the course of the simulations using Sim-alpha. Simulations were carried out together with John and the code changes and the results were unified. He also pointed out the approach of simpoints for speeding up the simulations.

Professor Vishwani Agrawal has taught quite a few of my courses and every bit of information I gathered from his classes have been extremely helpful. I would also like to thank him for recommending me for a summer internship at Texas Instruments (TI) in India. The hardware design and test concepts I eventually learnt at TI are invaluable. Professor Nelson's course on EDA tools, Professor Singh's course on VLSI design and testing and Professor Reeves's class on Digital Signal Processing require special mentioning as they were highly informative and helped me identify my interests in electrical engineering.

I am grateful to my friends here at Auburn who made my sojourn an eventful and memorable one. I would like to specially thank my friends Santosh Kulkarni and Pratap Prasad for their advice and support during tough times and for gleefully sharing my good times. I am thankful to Balapradeep Gadamsetti for being a great buddy. I cannot end without thanking Dr Dave Sree and his family for giving glimpses of India at Auburn.

My experiences at Wipro Technologies Ltd have helped me approach the challenges at graduate school in a more mature fashion. I am thankful to my teammates at Wipro for creating a great team based environment. The internship at TI Ltd further enhanced and polished my knowledge in the area of VLSI after my coursework at Auburn laid the foundation. I am grateful to my manager and my mentor at TI for providing an excellent research environment.

No amount of gratitude can ever be enough towards my parents Venkatesh T. L and Sathyavathi G. S and brother Rohith, who have been there for me every moment. I wouldn’t have achieved anything without their constant love and emotional support.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

1. SRAM - Static Random Access Memory.
2. DRAM - Dynamic Random Access Memory.
3. CPU - Central Processing Unit.
4. L1/L2 - Level 1/Level 2 Cache.
5. LRU - Least Recently Used.
6. SPEC - Standard Performance Evaluation Corporation.
7. I/O - Input/Output.
8. DMA - Direct Memory Access.
9. GPU - Graphics Processing Unit.
10. ILP - Instruction Level Parallelism.
11. ALU - Arithmetic and Logic Unit.
12. PC - Personal Computer.
13. DCA - Direct Cache Access.
14. NIC - Network Interface Controller.
15. IOC - I/O Controller.
16. MC - Memory Controller.
17. MSHR - Miss Status Holding Register.
18. LLC - Last Level Cache.
19. SDRAM - Synchronous DRAM.

Chapter 1. Introduction

Computer designs and related technologies have made incredible progress in the last half century. There has been a constant scaling up in speed and scaling down in size every generation. This can be strongly attributed to the advances in semiconductor devices and also to innovative designs at the architectural level. It has also given rise to the notion that *smaller is faster*. Memory and computational logic are analogous to cement and water when it comes to the construction of a computer. There are three digital circuit implementation factors critical to the design of the state of the art computer, which scale fast but at different rates relative to each other. Integrated circuit logic technologies, semiconductor memories and magnetic disk technologies form those three main components of a computer in the decreasing order of speed.

Circuit logic density scaling has always followed the Moore's law [1] with the transistor count doubling every 1.5 years. Memory modules such as Register files, Static Random Access Memories (SRAMs, present on chip) and Dynamic RAMs (present off chip), have also increased in capacity at the same rate due to the transistor device scaling. But large interconnect capacitances have resulted in slower access speed for larger memory units. This explains the speed gap between the memory devices and the logic circuitry. Disk density has been improving by 50% per year, almost quadrupling in three years. Since disks have mechanical parts, they can never match the speeds of the RAMs. Hence they are mainly used for mass storage. As a consequence of this varied rate of scaling the speed gap between memory devices and computational logic has been widening, thereby creating several performance bottlenecks. Memory hierarchies are used in order to bridge this gap between these three component levels and ease the constraints.

This chapter introduces the typical cache and memory setup in modern day processors and the performance issues associated with them. The chapter concludes with a description of the problem addressed in this work.

1.1 Caches and Memory Hierarchy

A cache is nothing but a small memory unit that stores data for future data requests to be serviced faster. The keyword here is *small*, because a smaller memory structure would have lower access latencies. A typical memory hierarchy in present day processors, both single and multi core ones, starts with the register files within the processor core and gradually moves towards larger but slower memory levels comprising expensive cache memories and ends in either the disk or network storage elements. The main motive behind this arrangement is to bridge the speed gap between the Central Processing Unit (CPU) core and the slower memory devices. This is very clearly illustrated in the Figure 1 reproduced from [2] below.

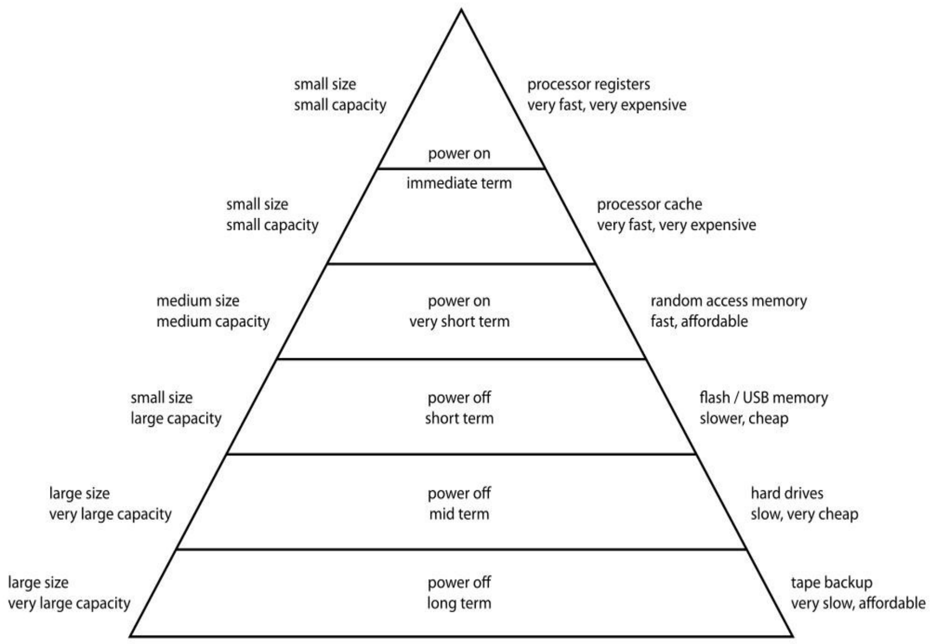

Figure 1: Memory hierarchy in a computer

A successful cache access is termed as a hit and failure is called a miss. This applies to both reads and writes. A read hit occurs when the requested block is present in the cache and a miss occurs when it isn't, prompting an access to the next level of cache with greater access latency. Similarly a write miss occurs when the modified data could not be successfully written to the next level of the cache because of the data block being absent from the cache. Let us consider a 3 level memory hierarchy comprising of a level 1 (L1) cache, a L2 cache and the memory for an example. If the probability of a hit in a level i memory structure is given by h_i and if T_i is the access time in cycles for the corresponding cache level, the average memory access time in cycles is given by this expression and provides a good performance measure,

$$T_{Average} = T_1 + (1 - h_1)[\ T_2 + (1 - h_2)\ T_3] \ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ (1)$$

Miss rate $(1 - h_i)$ reduction is the primary motive behind all cache based designs. Designs that do not address large miss rates would essentially lead to more program stalls and a smaller processor throughput even with pipelined and superscalar architectures. The 'temporal' and 'spatial' locality of the cache blocks are used for mitigating the miss rates in caches. Programs vary widely in terms of workloads, algorithmic complexity and size. Hence, it is also hard to design a cache hierarchy that suits perfectly for every program. However, we can always design one for optimal performance requirements by analyzing the tradeoffs involved. Memory hierarchy design is simpler for a set of applications that have similar and fixed workloads.

1.2 Bottlenecks and Tradeoffs in Cache Design

Memory hierarchies are very much required for cushioning the impact of access latencies due to slower devices, but a certain amount of tradeoffs are required for an optimal design. Reducing the number of cache misses has been the primary goal of most designers as it addresses both miss rate and miss penalty. Some of the basic cache design methodologies are listed below:

1. As seen from equation 1, the miss rate greatly affects the average memory access time. To reduce the miss rate caches (cache memories) with larger block sizes are used. As a drawback, a larger block size increases miss penalty after a certain optimal value since it would consume more cycles to transfer a block from the memory.
2. Larger caches certainly help in reducing miss rate, but the miss penalty increases as it takes more cycles to access a larger memory device. Caches with multiple banks are a good option if the data sets of the programs are large.
3. Using a higher associativity cache also helps in reducing the miss rate. This reduces the number of conflict misses. But the hardware complexity of the data retrieval circuitry increases because we now have to select between multiple 'ways'.
4. Processors typically use two levels of caches. By increasing the number of levels to three, we can get some speed-up.
5. Miss penalty can be reduced by giving more priority to reads than writes. In a setup with write buffers, on a miss we can check the buffer for the requested block. Writing the block to the memory and then reading it back would add to the miss penalty.

The tradeoffs between the hardware overhead and speed with caches are quite clear now. In addition to these, the write traffic handling is a major task for the cache controller. In programs involving large workloads, almost every cache miss results in an eviction as there is never much space on the cache for the incoming block. Write buffers are quintessential to every cache for absorbing the write latency (discussed in later chapters), and they are not foolproof either. Write buffer induced processor stalls can be attributed to the following three reasons [3]:

1. Full stalls occur when the buffer is full. The processor would have to retire the entries in the buffer to make space for the replaced cache entry causing stalls as the requested block has to wait.
2. A read-access stall occurs when a read miss in L2 cache encounters a delay in reading from the

memory because the write-back buffer is currently writing to memory.

3. A read-hazard stall occurs when L2 read miss finds its data in write-back buffer. However, this hazard can be avoided if write-back buffer entries and L2 cache entries can be swapped.

There are several strategies for cache write handling. Designs explained in [3], [5] and [6] have shown that write buffers contribute significantly in mitigating stalls. Jouppi in [7] and [8] proposed the victim buffer for handling conflict misses that mainly occur with direct mapped L1 caches. Chu and Gottipati [3] examine various factors to be considered for write buffer performance evaluation in their work. They find that even a single word of buffering yields a substantial gain in performance. Write buffer strategies are deeply analyzed in [9]. Having a deeper buffer provides more write merging opportunities and also reduces conflict misses. A read bypassing strategy, mentioned earlier, helps in holding the write data until the read takes place. An eager writeback strategy helps in balancing the accesses on the main system bus to reduce delays due to bus contention by committing Least Recently Used (LRU) blocks to memory earlier than the expected time. Handling the write traffic in such a way that there is complete concurrency between reads and writes will act as the upper bound on any improvement that can be achieved by addressing the write buffer issues.

1.3 Problem Description

Write data traffic to memory constitutes up to 30% of the total communication traffic to the memory in most of the modern computer configurations and with many of the existing software programs. The results shown in Figure 2 convey the same with some of the Standard Performance Evaluation Corporation (SPEC) CPU benchmarks for a typical uniprocessor architecture operating at 3GHz and having a 2MB on chip L2 cache using Sim-alpha, an Alpha 21264 processor simulator. In Figure 2, our simulations with SPEC benchmarks show that as much as 30% of the traffic between the CPU and memory is comprised of writes.

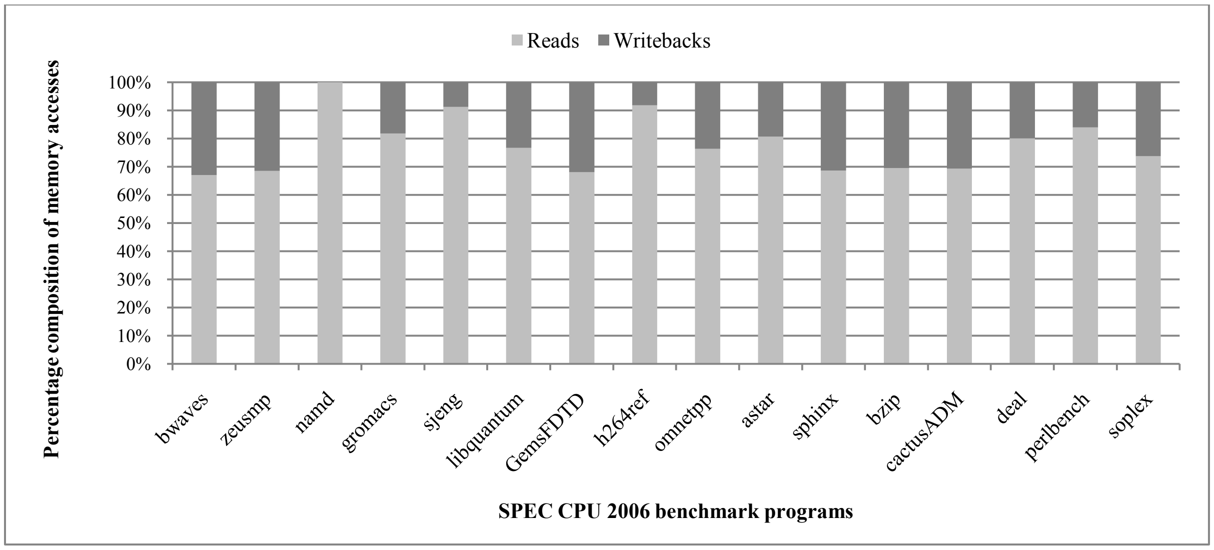

Figure 2: Memory access requests per 100 million instructions

The amount of queued cycles per request on the main system bus that results due to the access conflict between the read requests and the write commits from the memory whenever the write buffer becomes full. Write intensive benchmarks have shown that write buffer induced stalls can add significant latency to read misses in the last level cache. Research work in the area of write buffer analysis is minimal, but the works in [4], [5], [6] and [9] agree that write buffers do contribute to processor stalls, a case of the solution itself becoming a problem. The average number of cycles required per instruction execution is significantly lower than the average queued cycles per request on the main bus. This does not mean that the program execution is entirely blocked because of the queuing delay (design features such as 'out of order execution', 'speculative execution' and 'non blocking caches' ensure this does not happen), but there is certainly a major impact on the program execution time. This indicates that those instructions that have to endure the penalty of L2 cache misses take a large beating in execution time because of the conflict between the incoming reads and the outgoing write traffic.

The above mentioned setbacks with write buffers are addressed by using a hardware enhancement and a write-back strategy to support the main system bus. This architecture along with I/O techniques such as ‘direct cache injection’, ‘memory mapped I/O’ and ‘interrupt driven I/O’ which communicate directly with the CPU cache (as opposed to DMA) can make the best use of the main bus by efficient memory bandwidth utilization. The proposal is to have a dedicated bus, smaller in bandwidth to the main bus, to handle cache writes to the memory. Having a separate bus will also help I/O communications and Write-backs to happen in parallel in the case of the above mentioned techniques. The benefit of using a secondary bus to handle all of the cache writes to the memory has been studied in this thesis on some of the latest SPEC benchmark programs. Serial bus speeds are shown to be enough to handle the write traffic and be used as the secondary bus. A secondary bus controller that snoops for main bus traffic and determines the cycles best suited for a writeback to the memory is the hardware addition required to allow the link to function in the presence of I/O traffic and read requests. The main idea is to have write buffer entries retire ahead of time and only during those cycles where the main system bus is free from either a communication with the memory or an I/O device. One such arrangement has been extensively simulated in this work for different I/O data rates and also for Direct Memory Access (DMA) and Direct I/O transfer techniques.

Chapter 2. Background on Processor Architecture

Processors are classified into various different categories based on the architectural design. They can be categorized based on the instruction set complexity, number of cores, internal register length and number of threads per core to name a few. One thing that is common to all these designs is the caching of data for faster access. Almost every processor has multiple levels of cache and they use a common system bus to communicate with the memory and other peripheral devices like Graphics Processing Units (GPUs), I/O devices and Network Interfaces. This chapter throws light on a typical uniprocessor architecture that has later been used for simulations in this work and also provides an insight into the concept of multi core processors.

2.1 Uniprocessor and Multiprocessor Architectures

A typical uniprocessor is made up of only one processing core which can in turn have a pipelined and/or superscalar architecture that make use of instruction level parallelism (ILP). The former executes almost one instruction per cycle by having pipeline registers store the control information while a superscalar architecture incorporates multiple processing resources to enable two or more instructions to execute in parallel. A lot of the present day uniprocessors incorporate both pipelining and superscalar features to extract the best possible performance. Also, the architecture that exploits ILP in the best possible way is the one that guarantees a good instruction throughput. Several hardware-software techniques are used to make this happen. Resources can be register files, ALUs, branch predictors and multipliers to name a few. Resource redundancy can help us run multiple threads in parallel on the processor thus resulting in faster program execution. Typical uniprocessor architecture from [10] is shown in Figure 3.

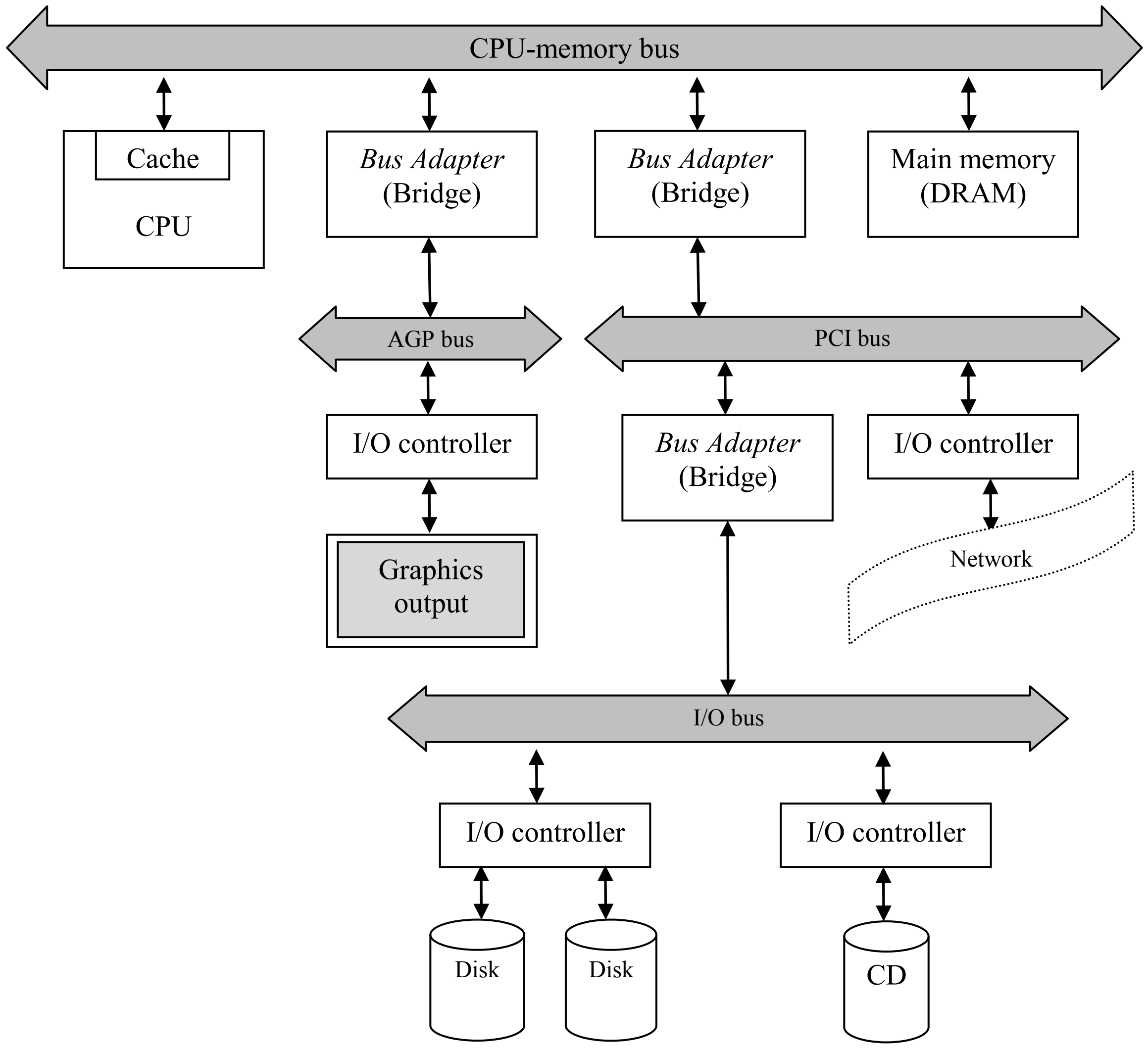

Figure 3: A typical bus based computer architecture [10]

Multi-core architectures are in vogue today because the scalability of uniprocessors has reached its limits and researchers have leveraged on the concepts of parallel programming. This has led to the use of several processor cores of reasonable speeds to perform the task and make use of the multiple threads in programs in a more efficient manner. These architectures exploit both instruction level and thread level parallelism. Figure 4 shows a simple multi-core processor block diagram. It is a common design practice for each processor core to have a local L1 cache and a shared L2. An interconnection network between the L1 and L2 caches handles data transfers between the two. A cache coherency protocol checks for inconsistencies between the two levels and the memory. The L2 cache can later be connected to the main memory through a main system bus.

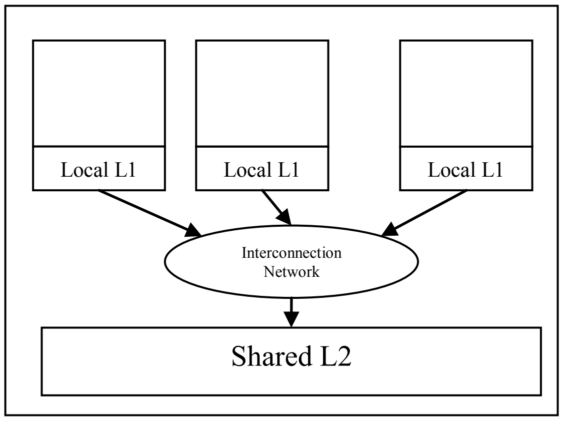

Figure 4: A simple multi core processor architecture

2.2 Processor System Bus

In a computer system, the various subsystems will have communication interfaces to each other. For instance, the CPU needs to communicate with the Memory and also with the I/O devices because the executing program comprises of both memory and I/O bound instructions. This communication is commonly done using a bus. The bus serves as a shared communication link between the subsystems. The two major advantages of a bus based system are low implementation cost and versatility. By defining a single interconnection scheme, new devices can be added easily and peripherals can even be moved between computer systems that use a common bus. The cost of a bus is low because a single set of wires is shared among multiple devices. One major drawback with a bus is that it creates a communication bottleneck especially when there is I/O traffic on the bus along with the regular memory traffic. In server systems where I/O is frequent, designing a bus system capable of meeting the demands of the processor is a major challenge.

One of the main challenges designers face with a bus based design is that the maximum bus speed is largely limited by physical factors like the length of the bus and the bus loading (number of devices on the bus). The desire for high I/O rates and high I/O throughput can also lead to conflicting design requirements. Buses are traditionally grouped into CPU-Memory buses (main system bus) or the I/O buses. I/O buses may be lengthy, may have many types of devices connected to them, have a wide range in the data bandwidth of the devices connected to them, and normally follow a bus standard. CPU-memory buses, on the other hand, are smaller in length and faster. Several bus bridges are used to connect the buses of different bandwidth and speed specifications. Ultimately, all of the I/O buses connect to the main system bus as shown in Figure 3.

Any communication over the bus happens between a master, who initiates the transaction and the slave who services accordingly. A situation with multiple masters on the bus would call for some kind of an arbitration mechanism. Table 1 illustrates the cost and performance trade-offs that need to be looked into while choosing a bus design. One thing that is clear from the table is that 'higher performance comes at a cost'. The first four points are self explanatory. It also talks about *split transactions* and how they aid in performance at a higher cost. The idea behind *split transactions* is to divide bus events into requests and replies, so that the bus bandwidth can be utilized in the time between the request and the reply.

Table 1: Main trade-offs for a bus design [10].

Option	***High Performance***	***Low Cost***
Bus Width	Separate address and data lines.	Multiplex address and data lines.
Data width	Wider is faster.	Narrower is cheaper.
Transfer Size	Multiple Words have less bus overhead.	Single-word transfer is simpler.
Bus masters	Multiple entities.	Single master requires no arbitration.
Split transaction	Yes - separate request and reply packets get higher bandwidth.	No - continuous connection is cheaper and has lower latency.
Clocking	Synchronous	Asynchronous

The final item in Table 1 is about bus clocking and it is concerns whether a bus is synchronous or asynchronous. If a bus is synchronous, it includes a clock in the control lines and a fixed protocol for sending address and data relative to the clock. Since little or no logic is needed to decide what to do next, these buses can be both fast and inexpensive. Major disadvantages include clock skew problems, which limit the length of the bus and a fixed clock rate means that everything on the bus must run at the same pace. Asynchronous buses, on the other hand, are not clocked. Instead, self-timed, handshaking protocols are used between the bus sender and receiver. It is much easier to accommodate a variety of devices and to lengthen the bus without worrying about clock skew. This comes at the cost of increased traffic on the bus causing large queuing delays for other traffic. It is not surprising to see the CPU-memory bus to be synchronous and an asynchronous I/O bus in computer architecture.

2.3 Input/Output techniques in modern day computers

As explained in the previous chapters, I/O refers to the exchange of data between the CPU and the peripheral devices. Traditional as well as some current techniques under research are discussed in this section. Table 2 lists the peak bandwidths which some of the fastest I/O buses are capable of. Though these are just the maximum possible numbers and the I/O traffic may not always attend such high rates, it gives an insight into the potential traffic that can be associated with I/O. One more point to note is that most of this traffic transits via the main system bus (CPU-memory bus) before reaching its destination (see Figure 3). This destination for all practical purposes is either the CPU or the memory.

2.4 Memory mapped I/O

In this type of I/O a peripheral device is connected to the CPU's address and data lines exactly like memory through some mapping, so whenever the CPU reads or writes to the address associated with the peripheral device, the CPU transfers data to or from the device. This mechanism has several benefits and only a few disadvantages. The prime advantage of a memory-mapped I/O subsystem is that the CPU can use any instruction that accesses memory to transfer data between the CPU and a memory-mapped

I/O device. The MOV instruction is the one most commonly used to send and receive data from a memory-mapped I/O device, but any instruction that reads or writes data in memory is also legal.

Table 2: Contemporary I/O bus bandwidths.

Bus Name	***Peak Bandwidth (GB/Sec)***
SATA 3.0 [11]	0.750
Light Peak [12]	1.25
USB 3.0 [13]	6.25
PCI Express 2.0 [14]	2 - 16
AGP [15]	2.133
QPI [16]	19.2 – 25.6
HyperTransport [17]	22.4 – 51.2
10 Gigabit Ethernet (10GBASE-X) [18]	1.25
40 Gigabit Ethernet (40GBASE-X)	5
100 Gigabit Ethernet (100GBASE-X)	12.5
Infiniband (SDR, 12X) [14]	3
Infiniband (DDR, 12X)	6
Infiniband (QDR, 12X)	12

A major disadvantage of memory-mapped I/O devices is that they consume addresses in the memory map. Generally, the minimum amount of space that can be allocated to a peripheral (or block of related peripherals) is a four kilobyte page. Therefore, a few independent peripherals can wind up consuming a fair amount of the physical address space. Fortunately, a typical Personal Computer (PC) has only a couple dozen such devices, so this isn't much of a problem. However, some devices, like video cards, consume a large chunk of the address space (e.g., some video cards have 32 megabytes of on-board memory that they map into the memory address space).

2.5 Interrupt driven I/O

In the case of programmed I/O, the CPU is busy waiting for an I/O opportunity and as a result remains tied up with that I/O operation until it is completed. This disadvantage can be overcome by means of interrupt driven I/O. In Programmed I/O, CPU itself checks for an I/O opportunity, but here the I/O controller interrupts the execution of CPU whenever an I/O device wants to initiate a transaction. This way the CPU can perform other computations in the mean time and execute an interrupt service routine only when an I/O operation is required, which is quite an optimal technique. A priority scheme determines what happens in the case of simultaneous interrupts. A *fixed* priority scheme results in devices getting assigned priorities in a fixed order. This may result in some low priority devices not being serviced enough. A solution to this is to assign priorities in a *rotational* order. This scheme rotates the highest priority among all devices by shifting the priorities.

2.6 Direct Memory Access (DMA)

DMA technology provides special channels for CPU and I/O devices to exchange I/O data, and the memory is used for buffering the I/O data. When the CPU wants to handle I/O data, it triggers the DMA write operations that transfer the I/O data from I/O devices to the memory. On the opposite direction, when the CPU writes data to I/O devices, the DMA read operations (transferring I/O data from the memory to I/O devices) are performed.

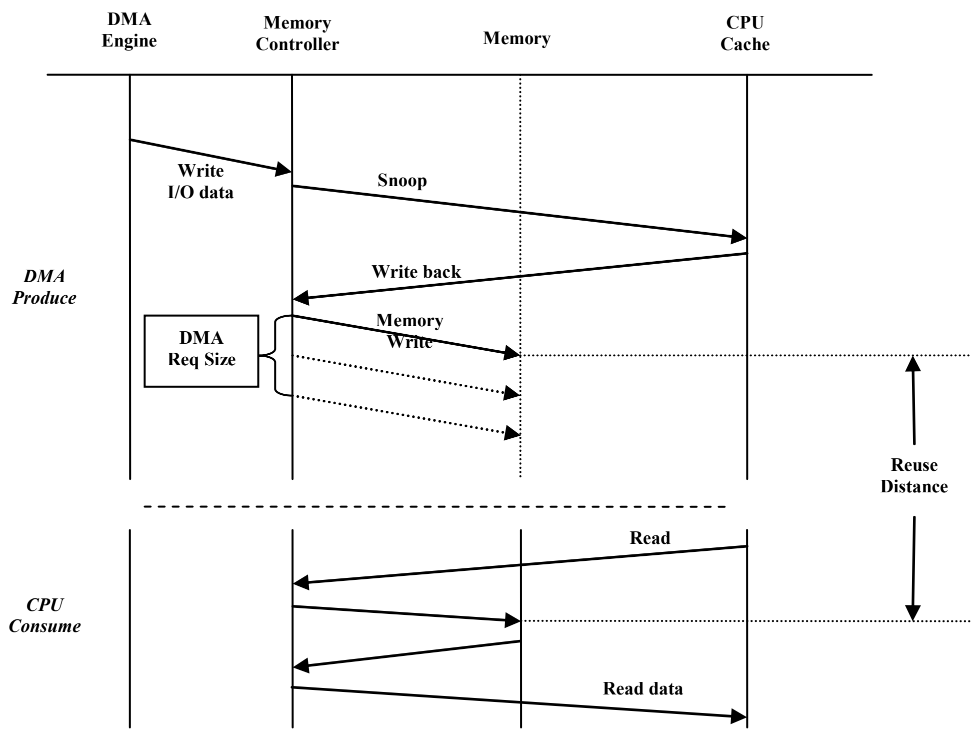

Figure 5: DMA flow diagram

The data flow diagram for a DMA transaction over different levels of the memory hierarchy is shown in Figure 5 for a DMA produce - CPU consume direction, reproduced from [19]. The processor, memory and the DMA engine are involved in the interactions during a DMA operation. The interaction requires three data structures namely the DMA buffer, descriptor and destination buffer, all residing in the main memory. To start off, the device driver creates a descriptor for a DMA buffer. The driver allocates a DMA buffer in the memory and initializes the descriptor with the DMA buffer's start address, size and status information. The driver informs the DMA engine of the descriptor's start address. DMA engine then loads the descriptor's content from the memory. With the DMA buffer's start address and size information extracted from the descriptor, the DMA engine receives the data from the I/O device and writes the data to the DMA buffer. After all I/O data is stored in the DMA buffer, the owner status of the descriptor is modified to be the DMA engine. The DMA engine sends an interrupt to the processor to indicate the completion of the receiving operation. The driver handles the interrupt raised by the DMA engine and copies the received I/O data from the DMA buffer to the Destination buffer. Then, it frees the DMA buffer. The processors adopt snooping-cache scheme for maintaining I/O data's coherence, accordingly they need to send snoop requests to the processor's data cache to invalidate those cache blocks that pertain to the I/O data under DMA request. Consequently, when the CPU consumes the I/O data, the compulsory misses will take place and trigger the memory read requests to the memory controller.

2.7 Direct Cache Access (DCA) and Cache Injection

In addition to the traditional techniques discussed above, DCA and cache injection are two other techniques that attempt to ease the memory bottleneck but letting the I/O device directly inject I/O data into the processor's cache. These techniques are producer driven when compared to the previously discussed techniques such as DMA, Interrupt Driven I/O and programmed I/O, which are consumer driven. Both of them are well suited for the large data rate network I/O over the Gigabit Ethernet. DCA [20] is basically a cache coherency optimization that delivers inbound data from a network interface controller (NIC) directly into processor caches dramatically reducing stalls due to memory access of descriptor, packet header and packet payload data structures.

Another technique that is worth mentioning because it is one of the assumptions in the simulations carried out in our work is that of direct cache injection [21]. Cache injection addresses the continuing disparity between processor and memory speeds by placing data into a processor's cache directly from the I/O bus. This disparity adversely affects the performance of memory bound applications including certain scientific computations, encryption, image processing, and some graphics applications. As shown in Figure 6, reproduced from [21], the injection operation is first initiated by the NIC. Unlike in DMA, where the next step is to write to the memory, step 2 allocates incoming network data into the cache. If the processor uses this data promptly there is no need to fetch the data from the memory.

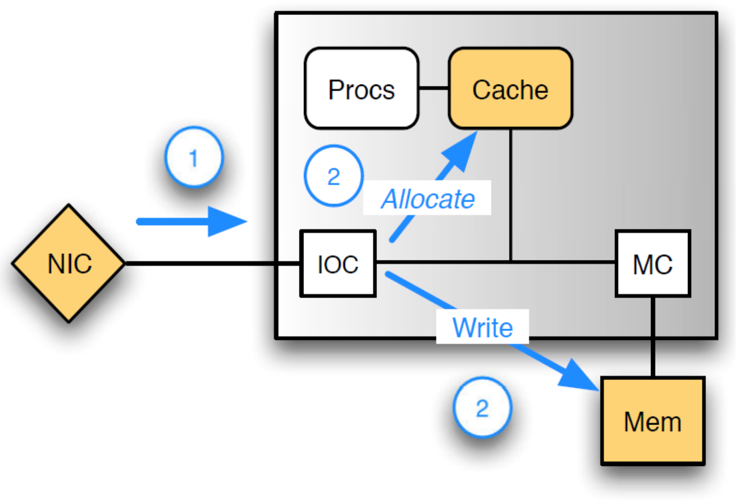

Figure 6: Direct cache injection based I/O.

2.8 Cache Writeback Strategies

Cache writeback, as explained before, is the process of committing data blocks back to the memory via the system bus. Several write buffering and writeback techniques are used to ease the memory access latency after a cache miss. The most basic ones of them all are the 'writeback' and the 'writethrough' techniques which are explained here.

a. Write Back

In this technique, the memory locations written are marked as dirty and are held in the cache until a read request evicts this line as a replacement. More often than not there is traffic towards the memory every time a cache read miss occurs some dirty cache line has to make way for the incoming datum. As a result a read miss in a writeback cache would require two memory accesses: one to retrieve the needed datum, and one to write replaced data from the cache to the store.

b. Write Through

When the system writes to a memory location that is currently held in cache, it writes the new information both to the appropriate cache line and the memory location itself at the same time. This type of caching provides worse performance than write-back, but is simpler to implement and has the advantage of internal consistency, because the cache is never out of sync with the memory the way it is with a write-back cache.

Writeback caches are more complex architectures than the ones using writethrough when it comes to implementation. Both the techniques as discussed later use some sort of buffering to absorb the impact of memory accesses. These methodologies are also used between the L1 and L2 caches. L1 caches usually comprise of separate partitions for instruction and data portions of the cache lines to aid in faster instruction fetch rates. Since writes can happen only on a datum, we can use a buffer only for the data cache among the two.

Chapter 3. Prior Work on Memory Hierarchy Optimization

Cache accesses and the penalties associated with them have been targeted by a lot of researchers over the past two decades. All the proposed cache write-back policies are aimed at minimizing the impact that a write to the next level would have on the processor pipeline. Consequently there have been a good number of innovative solutions such as Write Buffers, Victim Buffers, MSHRs, Eager retirement policies and Cache Prefetching. Most of the modern day processors use a good mix of all of these strategies. This chapter discusses a few of them.

3.1 Write Buffers

As mentioned in the previous chapter, cache write techniques used in uniprocessor architectures involve either the 'Write-Through' or the 'Write-Back' policy [10]. In a Write-Through technique, the modified data lines are written to the cache as well as the next lower level in the memory hierarchy. On the other hand, caches employing Write-Back usually mark the data line as 'dirty' to imply that the cache line is inconsistent with the next level in the hierarchy and do a write to the next level only when they are evicted by another incoming block. The disadvantage of Write-Through is that the processor has to stall since the memory is accessed every time there is a write operation. Write-Back also creates processor stalls whenever a dirty line is evicted from the cache and it has to be written to the memory to make space for the required line.

Using a 'write buffer' between the caches and the memory or the next lower level in the hierarchy helps in reducing this bottleneck. This is one of the earliest solutions proposed for tackling cache coherency and the associated latencies in the memory hierarchy. The cache writes directly go into the buffer than the next level and since the buffer has similar access latencies as the cache, we benefit through reduced stalls. Write buffers can also induce CPU stalls at times. Listed below are a few problems associated with write buffers [3]:

1. Full stalls occur when the buffer is full and the store cannot merge.
2. A read-access stall occurs when a read miss in L2 cache encounters a delay in reading from memory because the write-back buffer is currently writing to memory.
3. A load-hazard stall occurs when L2 read miss finds its data in write-back buffer. However, this hazard can be avoided if write-back buffer entries and L2 cache entries can be swapped.

There are certain occupancy based policies for retiring the buffer entries to the next level in the memory hierarchy. The buffer can retain a suitable number of entries for coalescing purposes, but can retire entries at the maximum possible rate when occupancy rises above a particular mark (number of valid entries in the buffer). Waiting until this mark before retiring means that sequential writes can achieve maximal coalescing. The most recently allocated entry cannot be retired until a new entry is allocated. We call the entry that triggers retirement, the high-water mark and name the retirement policy according to this mark. For example, a retire-at-2 policy would wait until 2 or more entries are valid in the buffer before starting the process.

Read access stalls on the other hand can be reduced by using an eager writeback policy, which is discussed later in this chapter. Flushing the write buffer on every load miss solves the load hazard problem, but at substantial cost. Techniques such as *Flush-full*, *Flush-partial* and *Flush-item only* are alternative solutions to this problem. *Flush-full* flushes the entire write buffer when the miss hits in the buffer. *Flush-partial* saves some work by flushing entries in FIFO order only as far as necessary to purge the hit entry. *Flush-itemonly* saves even more work by flushing only the hit entry. If a different entry is already being retired when the load hazard occurs, we assume this transaction completes first. Finally, read-from-WB allows the load miss to read its data directly from the write buffer without altering the buffer's contents, avoiding an access to the next level in the process [3], [9]. Deeper buffers also help in reducing the buffer full stall by storing more burst writes which is normally associated with the data intensive streaming programs. It also supports the concept of *lazy retirement*. As mentioned before, more contents held in the buffer provide more write merging opportunities and also improves the chances of a hit in the buffer upon a read miss in the cache. Figure 7 illustrates the writeback policy with a buffer.

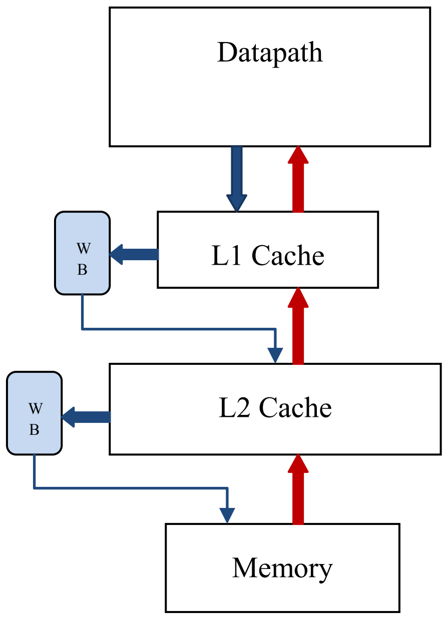

Figure 7: Write buffer example with write-back technique in a three level memory hierarchy.

3.2 Victim Buffers and Victim Caches

In cache terminology a 'victim' cache block is the one that is evicted upon a conflict cache miss. Many cache blocks get evicted in direct mapped caches during iterative function calls or context switches in a program. Since the probability of a cache block becoming a victim is very high in direct mapped caches compared to set associative ones, we need a buffer mechanism to hold these blocks as they may be required sooner in the program. So misses in the cache that hit in the victim cache provide a great chance to reduce miss penalty [7], [22], [23]. Experiments carried out on certain benchmark programs in [7] show that a small victim cache of 5 to 8 entries was enough to reduce the number of misses in a 1 to 4KB first level cache by about 80% of the cache misses.

The term ‘victim buffer’ on the other hand is associated with the buffering mechanism that ensures write merging. They are an advanced version of the write buffers discussed previously. A dirty cache line is buffered and any subsequent modifications to the line are merged to the entry in the buffer itself during write misses. Victim buffers are typically made up of more entries than a victim cache. Another difference between the two is in the fact that victim caches catch both ‘dirty’ and ‘non-dirty’ lines which are victims, whereas the victim buffer is usually meant for modified or ‘dirty’ cache lines [10]. The block diagram of a typical uniprocessor memory hierarchy can be seen in Figure 8. In a two cache arrangement, the L1 cache usually employs either a victim cache for support with a direct mapped cache or a victim buffer in case a cache with higher associativity is used. L1 employs the ‘write through’ policy. The L2 employs a write buffer and uses a writeback policy for coherence.

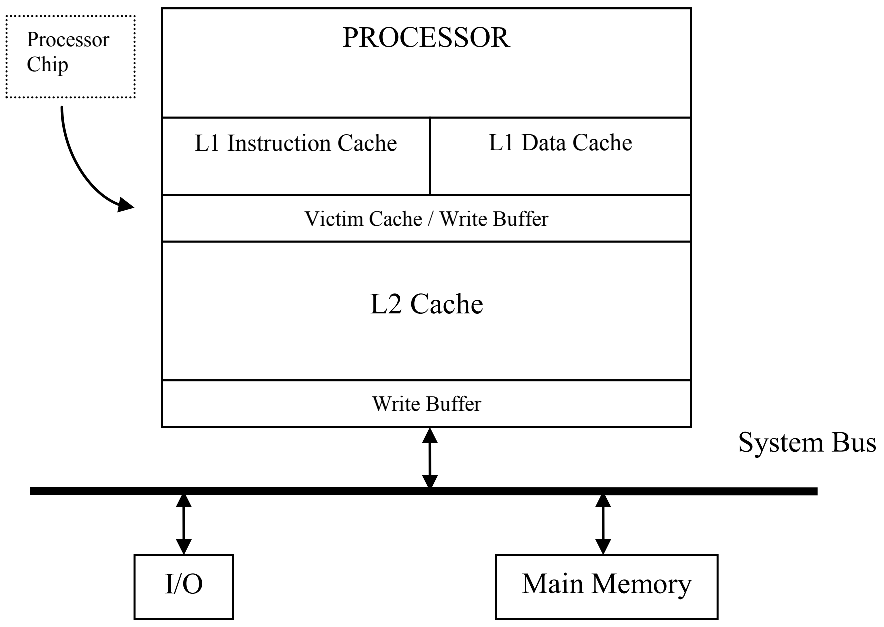

Figure 8: Processor Cache Architecture with Write Buffers.

3.3 Cache Prefetching

CPU cache prefetching involves fetching a block from the main memory into the CPU when the block has not been referenced in the expectation that it will be referenced in the near future. Hardware cache prefetching is specifically concerned with prefetching algorithms implemented solely by dedicated hardware without any software support. Two questions have to be answered before prefetching a block: which block to prefetch and when to prefetch. The simplest candidate to prefetch is the next sequential block after the one most recently referenced. This is illustrated in [24] with a technique called *always prefetch*. With this algorithm, every time there is a reference to block i, the cache is examined for block i + 1 (i.e., the next sequential block, in terms of ascending memory addresses). If block i + 1 is absent from the cache, it is prefetched.

A variation which requires fewer prefetches and prefetch lookups (i.e., look into the cache to see if the block is there) is called *prefetch on misses*, which prefetches the next sequential cache block if and only if the access to the current cache block is a miss. A more complicated scheme, known as *tagged prefetch* [24], keeps the number of prefetch lookups low while issuing more prefetches than prefetch on misses. In this case, each cache block has a single bit, called the tag, which is set to zero whenever the block does not reside in the cache. When a block is referenced by the processor, its tag will be set to one. A block brought into the cache by a prefetch, however, retains its tag of zero.

Figure 9 (reproduced from [24]) shows a typical hardware cache prefetch architecture. The two prefetch units, one for each cache, are responsible for issuing new prefetch requests to the main memory. During each clock cycle, each prefetch unit receives information like cache misses, cache hits, instruction types, and branch target addresses from the processor and the caches. Based on this information, it decides whether to issue a new prefetch request or not. If it does, the prefetch address is looked up in the corresponding cache. The request is issued in the next clock cycle if the data is not found in the cache. Issued requests from both prefetch units are not sent directly to the memory bus, though, but to a prefetch address buffer, each of which is organized as a FIFO queue with 16 entries. The oldest entry is sent to the memory bus only when the bus is free. If the buffer is full when a newly issued request arrives, the oldest entry is discarded from the buffer to make room for the new one. Whenever there is a cache miss, the address of the missing cache block is compared against every entry of the buffer. Any entry which matches the address represents a failed prefetch (because it is issued too late) and is discarded without being issued.

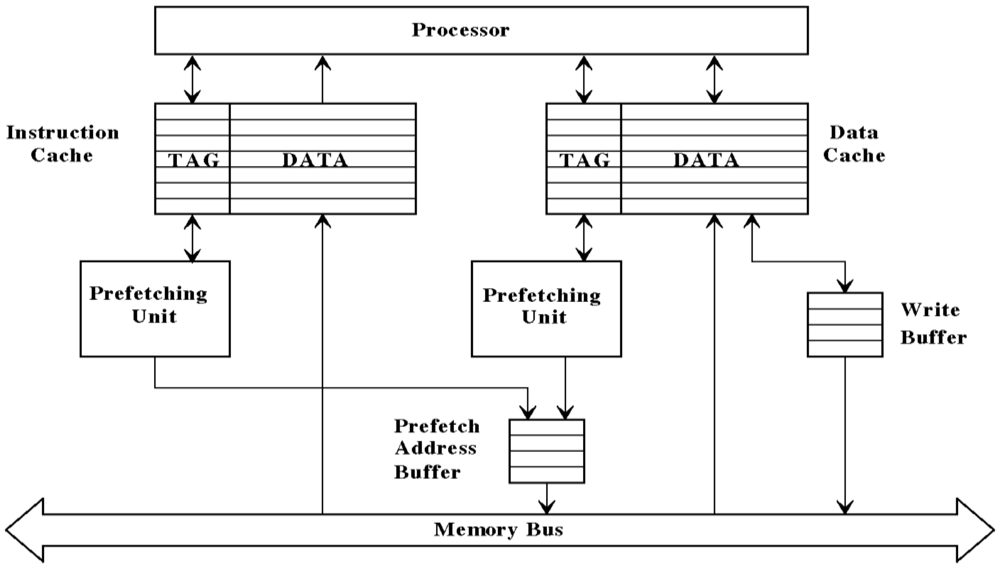

Figure 9: Cache Prefetching Architecture [25]

One of the disadvantages of cache prefetching is the unavoidable increase in memory traffic because of prefetches which are never referenced. The limitations of cache prefetching on a bus-based multiprocessor system are investigated in [26]. Results show that, when bus bandwidth is the bottleneck, prefetching will not improve performance, even when it reduces the demand miss ratio. Another disadvantage of cache prefetching is that useless prefetches may pollute cache contents by displacing useful cache blocks from the cache and, thus, cause new cache misses which would not have happened had there been no prefetching. All of these factors ensure the advantages from cache prefetching are minimal on CPU performance.

3.4 Miss Status Holding Registers

Out of order instruction execution is a popular technique in pipelined computers that allow a processor to fetch another instruction whenever there is a miss in the data cache. This means the processor need not wait until the data request is serviced by the next level in memory. A non blocking cache is used in such a situation to reap the benefits. Extra hardware is needed to store the cache miss information in the form of the requested address and that is where the Miss Status Holding Register (MSHR) comes into play. These registers hold the address information for the miss that is to be serviced. A *hit under miss* optimization reduces the effective miss penalty by helping during a miss instead of ignoring the requests of the processor [10].

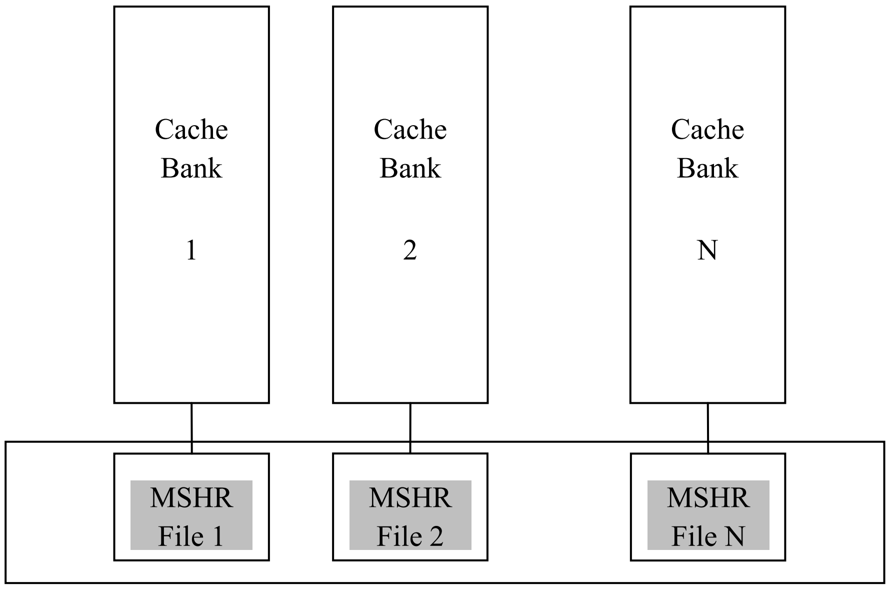

Figure 10: Miss Handling Architecture for multi bank caches

Most of the modern processors such as Intel Pentium 4 use multi banked L2 caches to facilitate parallel miss request servicing [27]. This optimization allows multiple misses over a miss, which means we can have multiple misses in queue. Conventionally each cache bank would have an MSHR file to facilitate storing of this miss information [28]. Figure 10 shows the block diagram of a typical multi bank cache and the MSHR arrangement.

3.5 Eager Writeback

The fundamental idea behind eager writeback strategies is to write the dirty cache lines to the next level in the hierarchy and clear the dirty bits earlier than in a conventional writeback. Since the main system bus handles a huge amount of data traffic for data intensive applications, this enhancement would be highly beneficial for performance improvement. The work in [29] explores one such technique by indirectly distributing the traffic on the main bus to make use of the idle bus cycles. The Eager writeback technique is a compromise between the writeback and write through techniques. Here the write commits are neither made upon every cache line modification like in write through nor does the write buffer waits to get filled. The dirty lines are evicted as and when the main system bus becomes free. However, the system bus carries more traffic than just the writebacks and reads from the cache hierarchy to the memory. Input/Output (I/O) traffic can cause a major bottleneck on the main system bus when network based I/O or other I/O intensive applications are running is considered. This technique does not provide much benefit with I/O considerations. In summary, the following two points highlight the shortcomings of the Eager writeback strategy:

1. The strategy does not 'snoop' the main bus to identify free memory cycles. This can be a hazardous in situations where clustered activity is present on the main bus. The write commits are just offset to an earlier stage through an early writeback of LRU lines with Eager writeback. This may not always solve the problem of bus contention.

2. During high I/O traffic densities on the main bus, the performance of the CPU decreases because of the large queuing for CPU read requests. This is a situation amplified by two kinds of traffic, CPU Produce-I/O Consume and I/O Produce-CPU Consume.

A majority of the processors today use an optimal mix of the above techniques to mitigate the memory access penalties that results from a cache miss. Some researchers also call for dynamic optimizations in order to optimize the performance for a particular type of application currently running on the processor.

3.6 Secondary Bus Architecture

The concept of a secondary bus to connect the level-2 cache to memory for cache writebacks has been explored in [30] by O'Farrell and Baskiyar. The simulations results on three data intensive micro-benchmarks show that adding an additional bus to support the main system bus would help in achieving significant reductions in queuing delays on the main bus. Such reductions in queuing delays offer superior temporal determinacy in a real-time environment. Their simulation results also compare well with 'free writeback' (it models a system in which dirty writebacks do not generate any memory traffic on the bus) and indicate that this bus can indeed parallelize reads and writes. It also discusses the feasibility of implementing such a bus as a serial or a wireless link.

Chapter 4. Secondary Bus Architecture

The main system bus is a bottleneck in bus based systems and an efficient use of the bus cycles calls for some kind of access control mechanisms. An additional bus can support by carrying the write traffic off the main system bus. Just adding a second bus would not help in easing congestion as it would only transfer the bottleneck to the memory interface from the L2 cache interface of the main bus. This write traffic would require a new write back policy (retirement policy), to make sure there is a less contentious path between the memory and the processor for reads. A bus controller is required for controlling who (write buffer, an I/O device or Read requests) gets access to the main memory and when. This chapter explains the design of the secondary bus architecture that is expected to ease queuing delay and result in improved program execution speed for the CPU. The secondary bus architecture was designed by Wang and Baskiyar [31].

4.1 Design of the Secondary Bus

The motive behind the secondary bus architecture design is to first provide a separate path from write buffers to main memory so that the three main reasons for CPU stalls (explained in Chapter 1) due to the main bus bottleneck are reduced. We refer to direct I/O here as a technique similar to cache injection or DCA (discussed in section 2.3.4), where the I/O data is directly written to or read from the processor cache. Direct I/O gives us an upper bound on the improvements possible with the architecture under discussion. Then, the following two time slices can be made use of to commit dirty cache lines in the write buffer to the memory over the main system bus:

a. In the absence of any data traffic directed towards or from the main memory.

b. During I/O transactions on the main system bus in the case of direct I/O.

The first condition is required because we are assuming a single ported memory here. A single port memory restricts the number of simultaneous accesses to just one, irrespective of whether it is for a read or a write operation. Hence, to return the cache lines in the buffer, choosing intervals when the memory is free eliminates queuing delays on the system bus as opposed to the techniques discussed in the previous chapter. Secondly, in the presence of any I/O traffic that is a result of a communication directly between the processor and the I/O device (as it happens during a Direct I/O transaction), the memory unit is available for write commits via the secondary bus. The design of their secondary bus based architecture [31] is shown in Figure 11 below.

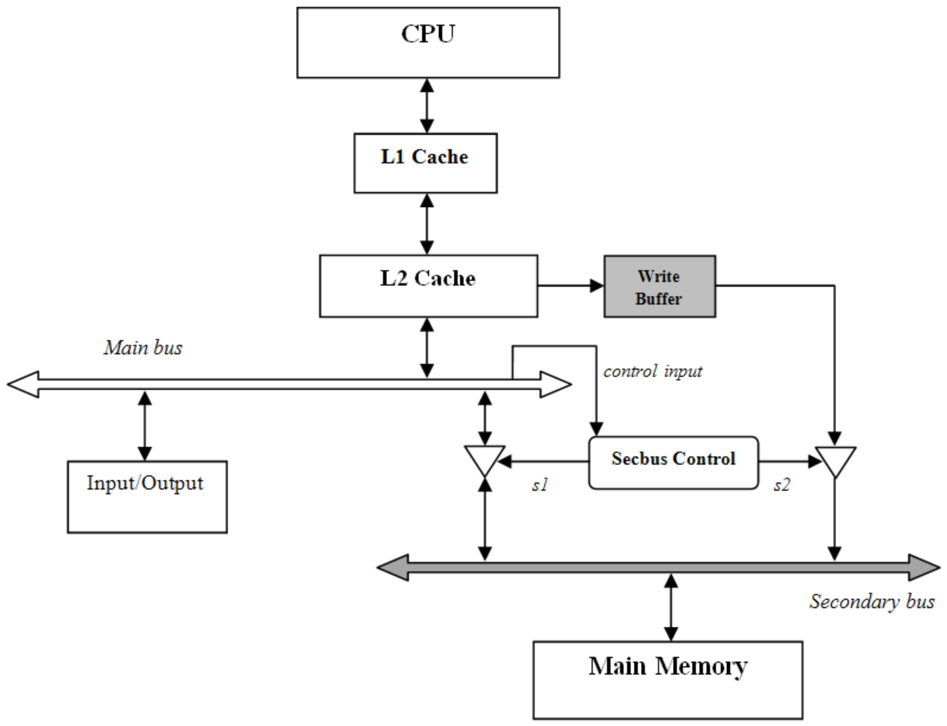

Figure 11: Architecture with the Secondary bus

As seen in Figure 11, the main bus is supported by the secondary bus during writebacks and I/O transactions. This is made possible by the secondary bus controller which does a snoop on the main bus and identifies the bus cycles where it is not busy with memory operations. These are the cycles where the secondary bus would take over and commit the dirty cache lines to the memory, giving a ‘faucet’ like control. The ‘control input’ to the secondary bus controller is made up of the main bus control lines that give information about the type of transaction happening on the bus. This can be address strobe, burst ready or the I/O control lines [32], which indicate when a transaction starts on main bus. The signals ‘s1’ and ‘s2’ are sent in accordance with the states of the main bus to arbitrate their memory accesses.

4.2 Design Issues

There are several design issues that need to be addressed while developing a write back policy for the secondary bus architecture. Some of them are explained here.

a. There can be situations when there can be read requests on the main bus at a point where there can be cache retirements happening on the secondary bus to the memory. One of the ways of handling this situation is to make the main bus request to queue up in the MSHRs and then complete the write operation until the writeback buffer is empty, while the other option would be to abort the burst writeback and enable the main bus to do the transaction with the memory. In this design, the main bus access is given priority so as not to contribute to the queuing delay on the bus. This creates a dependency for the secondary bus states on that of the main bus.
b. The addition of the secondary bus and the bus controller can consume some area on the motherboard and the memory controller hub respectively. Secondary bus though, is designed to be of a smaller width than the main bus and hence of a smaller bandwidth since it is only used for the writebacks. This would mean the transactions on the secondary bus would take longer than the main bus. This is not a concern because writes constitute a small percentage of the total memory traffic as discussed earlier and lesser bandwidth would suffice. However, the usefulness of the secondary bus depends on the amount of free memory bandwidth. If the main bus is busy with ‘direct I/O’ operations or idle for longer time, even a small secondary bus would be good enough to commit the dirty cache lines. This will lead to performance boost in situations when there is severe I/O traffic or clustered memory

accesses, which otherwise can lead to congestion on the main bus. This design ensures that the performance of a processor with a particular application will not be degraded as much as it would for a computer without the secondary bus.

c. In the case of direct I/O operations, as explained before we can parallelize the write back and I/O read operations. During DMA, the memory bandwidth is used up most of the time for one of these: I/O reads, I/O writes, CPU reads or CPU writes. This movement of I/O data between the processor cache and the memory adds more queuing delay to the traffic on the main bus, thereby creating a bottleneck. DMA reduces the number of available cycles for writeback. It would cause the write buffer induced CPU stalls to aggravate, which otherwise is one of the major areas of improvement with the secondary bus. So the secondary bus will work fine with DMA without any specific design modifications, but with reduced benefits.

Chapter 5. Simulation Setup for Performance Evaluation

Any new design or an enhancement to an older design always requires evaluation. Computer architecture research groups around the world use simulators like SimpleScalar, Sim-alpha, GEMS, SimOS and Simics to name a few. These simulators try to simulate the entire architecture along with the modification to our desired level of accuracy. Though the simulation data generated can never match the data collected from a real time run on an actual hardware for accuracy, they are faster and good for comparisons at an early design stage. In this work the Sim-alpha simulator [33] is used for the same purpose. SPEC CPU 2006 benchmarks are used for simulations as they a both CPU and memory intensive. The latter, especially, was very important as a good workout for the memory hierarchy is essential for secondary bus benefits to be visible. The changes made to the simulator encompasses the architectural design originally made in [31]. In other words, secondary bus architecture is very much suited for memory and I/O intensive programs. This chapter throws light on the simulation setups created using Sim-alpha and the SPEC benchmarks.

5.1 Sim-alpha Simulator

Sim-alpha is a validated, execution driven, Alpha 21264 processor simulator. It was written by extending the SimpleScalar tool suite [34]. Sim-alpha models the implementation constraints as well as the performance enhancing features of the Alpha 21264 processor. The simulator settings can be varied by the user to simulate the influence of parameters like cache sizes, memory speed, fetch width, issue queue sizes, bus bandwidths and many others associated with a computer.

The 21264 is a superscalar processor that can fetch and execute up to four instructions per cycle. It also features out of order execution, which results in critical path executions to start and complete quickly. It also has a branch prediction unit and executes speculatively. Coupled with high clock speeds this unique combination of out-of-order and speculatively execution, provide exceptional core computational performance [35]. The processor has seven pipeline stages as shown in Figure 12 (reproduced from [35]). Most of the present day complex applications cannot necessarily be run at a throughput of fours instructions per cycle. Some of them can take more than 1000 cycles to execute due to the access bottlenecks on the last level cache (LLC), the system bus and the memory. Though the numbers shown below in Table 3 are from the actual design of the 21264 processor, these can be varied using several flags or configuration files with sim-alpha.

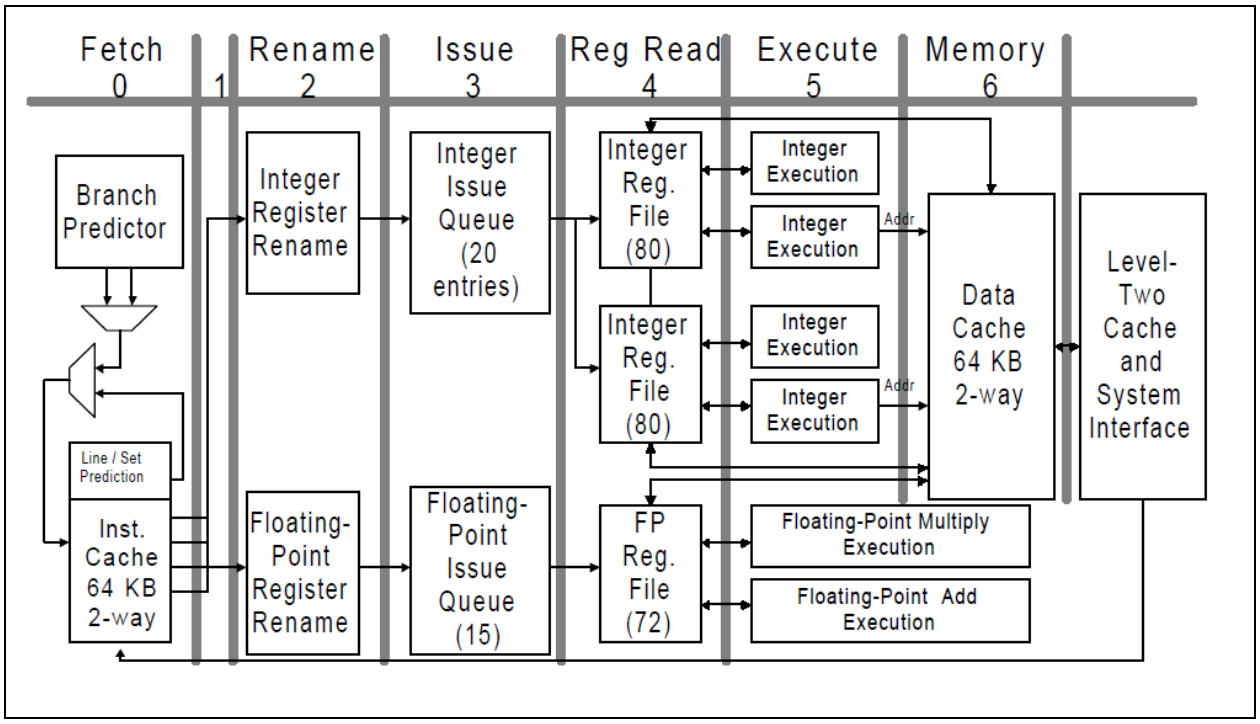

Figure 12: Microarchitecture of the Alpha 21264 processor [35].

Sim-alpha incorporates a detailed memory subsystem with support for multi level cache hierarchies, address translations, bus contention and a Synchronous DRAM (SDRAM) memory model. It builds on x86 machines and acts a cross architectural simulator, whereby it runs on a x86 machine and simulates binaries compiled for the Alpha 21264 instruction architecture. The average error by using sim-alpha as opposed to the actual Alpha processor is only about 2% as evaluated across a handful of micro benchmarks in [33].

5.2 SPEC Benchmark Programs and Simpoints

The SPEC CPU 2006 benchmarks suite [36] consists of integer and floating-point programs that represent a wide range of applications that we use today on our computers. These benchmarks are highly rated for evaluating several computer design components, mostly the CPU, memory subsystem and also compilers. Though some of the previous SPEC suites had problems exercising the memory subsystem either due to lack of working sets [29] or due to lesser application complexity, the 2006 programs run longer, have large working sets and are more complex. Video compression and speech recognition have also been added to these new benchmarks.

The 2006 suite of programs have a large amount of run times though its predecessors can now finish a run within minutes on the existing architectures. Runs times of the current benchmarks can range from machine weeks to months on a cycle accurate simulator like sim-alpha, before one can get access to the results. They also show high variability across several runs on the same set of data even after these accurate simulations. In order to ease this problem simulation points (simpoints) [37] are used during simulations. These small set of samples (simpoints) when simulated and weighted appropriately provide an accurate picture of the complete execution of the program with large reduction in the simulation time. Several days of run times are reduced to just a few hours with a slight compromise on accuracy. The methods used to extract these points and their weights are discussed in [37]. It also provides a list of files for the CPU 2006 group for quick use in simulations.

5.3 Sim-alpha Architectural Configurations Used in Simulations

Sim-alpha requires a processor and memory hierarchy configuration list to start a simulation run. We have used the data provided in Table 3 for simulations with the secondary bus. Modifications were made to the memory hierarchy with the addition of a write buffer to the last level cache. In the final structure, L1 cache had a victim buffer for support and the L2 (LLC) had the write-back buffer to handle write traffic. This setup made up the base architecture against which the secondary bus architecture would be compared later. Changes to the configuration file involved the addition of a new bus connecting the write buffer (added previously) to the memory, bypassing the main bus. It was made sure that the write traffic used only the secondary bus.

Table 3: Simalpha specifications

Processor Parameter	**Specifications**
Processor Speed	3 GHz
Level 1 Data Cache	8 way, 32KB, virtual-index virtual-tag
Level 1 Instruction Cache	8 way, 32KB, virtual-index virtual-tag
Level 2 Cache	8 way, 2MB, physical-index physical-tag
Number of MSHRs per Cache	8
Write Mechanism for Level 1 Cache	Victim Buffer, No Writeback Buffer
Write Mechanism for Level 2 Cache	Writeback Buffer, No Victim Buffer
Main Bus (Front Side Bus)	600MHz, 8B wide, 10 cycles of arbitration latency
Secondary Bus	600MHz, 1B wide, 10 cycles of arbitration latency

A major benefit of using such a secondary bus would be in those situations where I/O traffic uses a large part of the bandwidth on the main bus causing congestion for non I/O data. Hence, a simulation setup to generate I/O traffic at a rate described by a ‘Normal’ distribution was created. Normal distribution was assumed I/O traffic with the CPU because I/O can be mainly composed of network traffic and disk traffic. Figure 13 gives the probability density function used for our simulations. A dummy I/O device that would generate blocks of the size of cache lines was implemented and I/O injection frequencies were chosen to get a calculated bandwidth pinch. The I/O data rates tried were of 600 MB/Sec, 1.2 GB/Sec and 1.8 GB/Sec. These numbers were chosen based on the I/O bus bandwidth number shown in Table 2. I/O data eventually reside in the LLC by evicting some dirty cache lines in order to make space for this incoming data. These replacements can lead to some conflict misses later in the run and cause more traffic on the main system bus. Comparisons with Eager writeback [29] were also made possible with a different set of changes to the simulator. DMA I/O was discussed in section 2.3.3 and performance comparisons with the DMA I/O were also required. Simulator changes included implementing a DMA engine that would mimic the I/O injection and act as a bus master. This time I/O traffic went through the memory unit before landing in the LLC as opposed to Direct I/O, thereby reducing the memory bandwidth available for writeback.

Simpoints does help us in speeding up the simulations but faster simulations can only be achieved via a supercomputing environment. The Alabama Supercomputing Authority [38] provided us with a server cluster that was made up of some of the latest processors. As a result of multiple simpoints being run in parallel on the supercomputer nodes, the results for all of the 16 benchmarks were obtained in just a couple of weeks. Although separate such simulations had to be run for Eager writeback, each of the I/O injection rates, DMA I/O and Direct I/O conditions, each taking about two weeks, because of the inability of Sim-alpha to make use of multi processing.

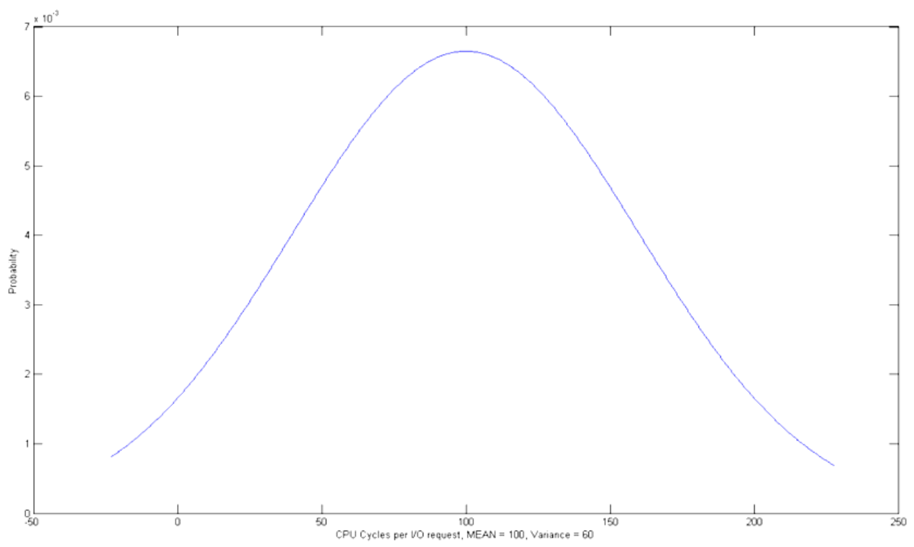

Figure 13: Probability density function used for I/O injection, Mean = 100 cycles and SD = 60 cycles.

Chapter 6. Simulation Results and Observations

In this chapter, the simulations results obtained with sim-alpha and their analysis are put down. The results shown include comparisons with the Eager writeback technique for all the different I/O frequencies, simulations with I/O traffic. The metrics used for evaluation included queuing delay on the main bus, maximum number of cycles taken by any instruction, average cycles per instruction execution and number of instructions taking more than 1000 cycles to complete. Though all of the metrics are interrelated, the results give understanding of their dependencies on each other. Reduction in some of the metrics like the average queuing delay for an instruction can be more beneficial to certain real time applications and systems as opposed to personal computers and server based systems.

6.1 Queuing Delay on the Main System Bus

The main bus in our simulations has a bandwidth of 4.8 GB / Sec and was shared by some I/O traffic (600MB / Sec, 1.2 GB / Sec and 1.8 GB / Sec) the writeback traffic and the read traffic. There are times when the bus is being used for servicing a number of clustered requests and another request that comes up at the same time has to be queued because of the limit on the number of outstanding requests. In real-time systems these queuing delays can become significant resulting in unexpected latencies and hard deadlines getting missed. Our simulations show significant reduction in queuing delays due to separation of the write traffic from the read traffic. Figure 14 shows the percentage queuing delay reduction achieved with the secondary bus against the base architecture for each of the SPEC programs at various I/O rates. Almost all of the programs showed great reduction in the queuing delays with an average reduction of nearly 99% during the absence of I/O traffic on the main bus. With I/O devices trying to communicate with the processor through the technique of direct I/O, we start to see reduced benefits, though Figure 14 conveys that the percentage reduction is still considerably high. It also indicates that a majority of the data queuing that occurs on the main bus is due to the write traffic in a write back cache and the benefit of using a secondary bus on queuing delay is very clear.

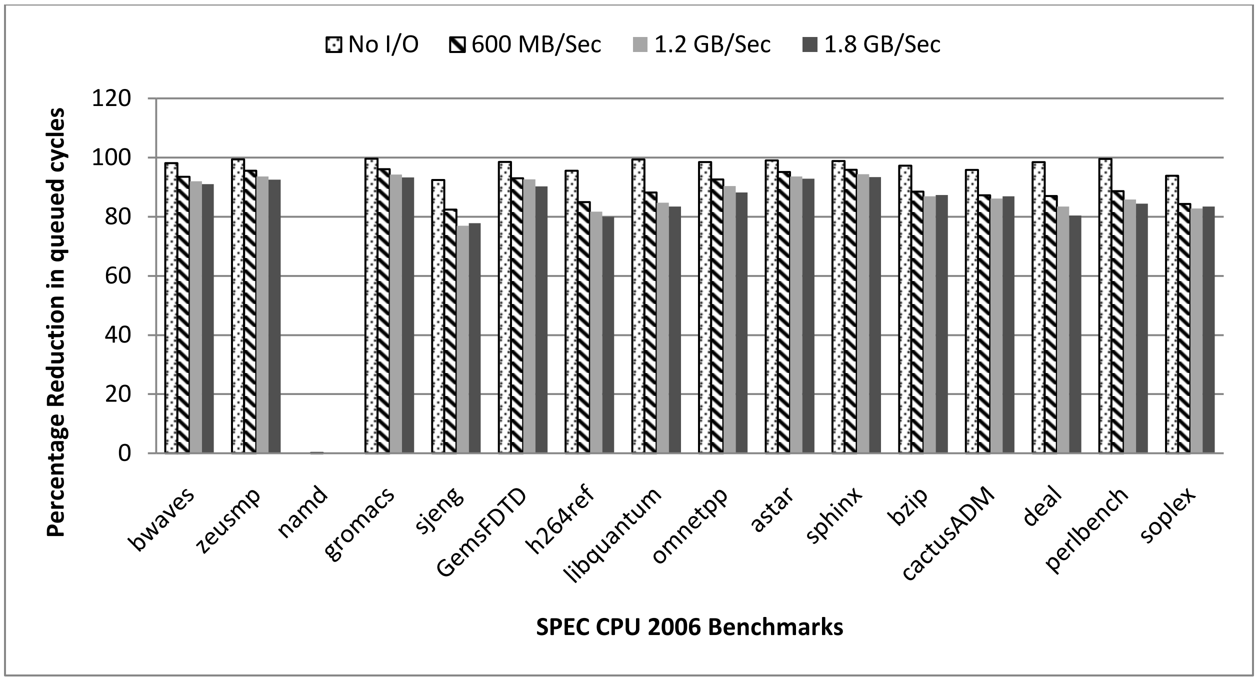

Figure 14: Percentage reduction in queuing delays across different I/O rates.

The comparisons with the Eager writeback and other types of I/O like DMA are shown in Figure 15. It is obvious that direct I/O extracts the best out of the secondary bus architecture when compared to DMA. When DMA is used, the I/O data travels through the memory before reaching the processor. This data is very much like any other read data from the memory, reducing the memory bandwidth for writeback using the secondary bus. With direct I/O, we can do a write back to the memory as well as an I/O read from the I/O controller in parallel. All these factors lead to a reduced performance for DMA with the secondary bus, but it is quite useful in reducing the delays nevertheless. Although Eager Writeback results in good reduction in queued cycles, it cannot match the advantage with a secondary bus. This is because the write back policy there does not snoop for free cycles on the main bus and only offsets the writes to an earlier stage. Figure 16 provides the total queued cycles information during an I/O rate of 1.2 GB/Sec.

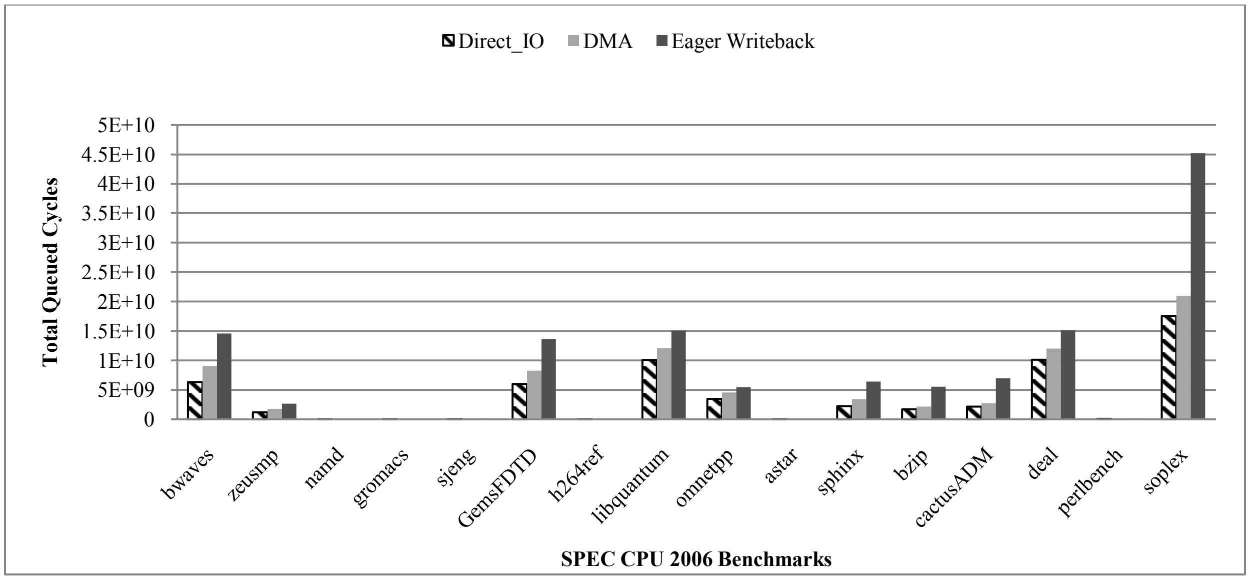

Figure 15: Total number of queued cycles during an I/O traffic rate of 1.8 GB/Sec.

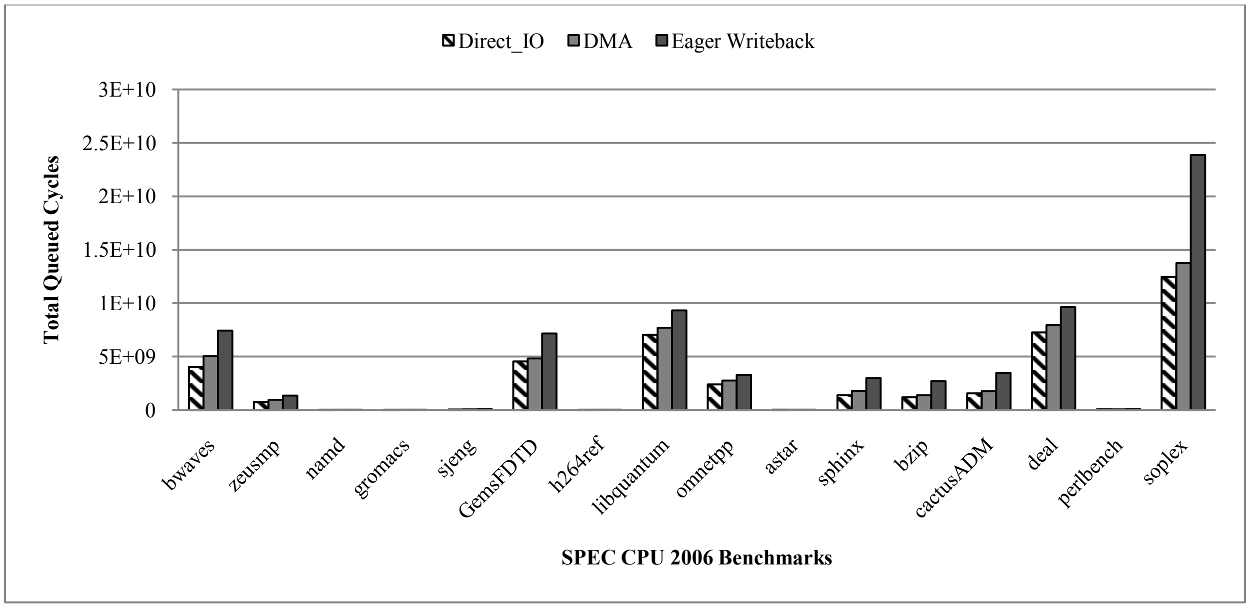

Figure 16: Total number of queued cycles during an I/O traffic rate of 1.2 GB/Sec.

6.2 Cycles per Instruction

A comparison of the 'cycles per instruction' between the secondary bus architecture and the base architecture, gives us the speed-up achieved. Figure 17 shows the percentage speed-up achieved across a range of programs from the SPEC CPU 2006 benchmark suite.

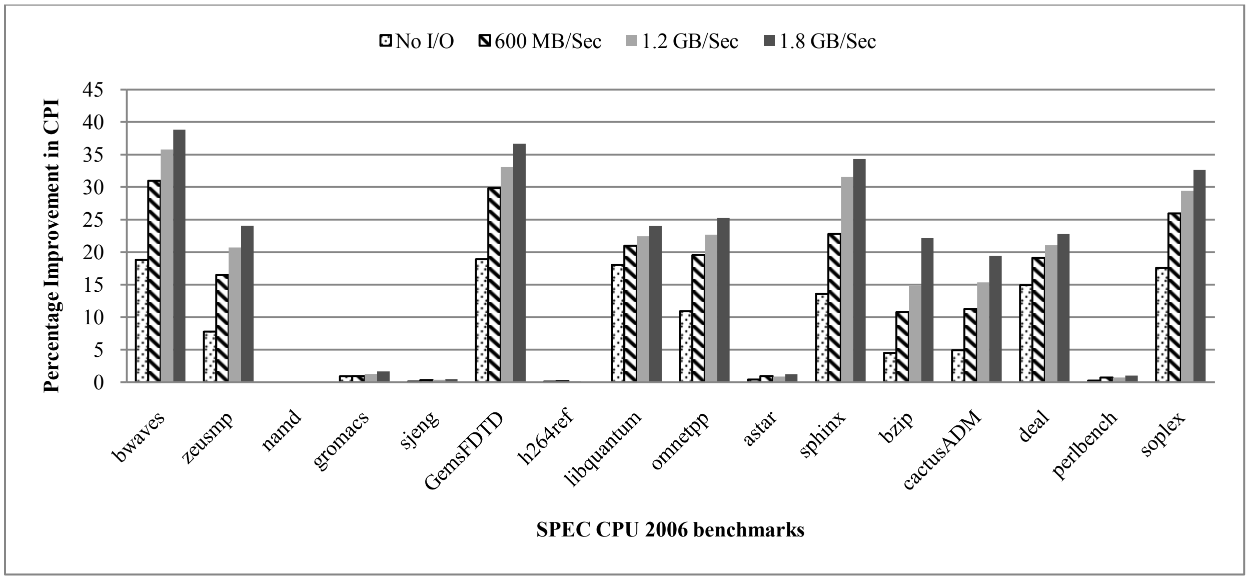

Figure 17: Percentage improvement in processor throughput with the secondary bus.

Speed-ups of up to 19% were achieved due to the addition of the secondary bus as seen in Figure 17 in the absence of I/O traffic on the main bus. With the presence of the secondary bus, read requests on main bus never waited for the dirty writeback traffic to be written to the main memory whenever it requested data due to a L2 cache miss. In the presence of I/O traffic further improvement was seen with speed-ups of up to 33%. In other words, the degradation of the base architecture was relatively severe when simulated with I/O traffic. The improvements in CPI are also a direct consequence of the reduction in queued cycles seen in the previous section. Programs that have a huge writeback rate or the ones that work on large data sets are the most beneficial of this architecture. On an average 13% speed-up over the base architecture was seen across 10 out of the 16 benchmarks we simulated with sim-alpha.

Results also show that the speed-up depends on how much the program strains the memory hierarchy. Processors using smaller second level caches lead to higher number of cache misses and hence writebacks. Thus programs having a very large working set could be more advantageous compared to the ones using smaller caches and working sets. This can be seen with programs like *namd*, *gromacs*, *sjeng*, *h264ref*, *etc*. The program *namd* did not have any writebacks to the memory and hence the architecture was never put to test during the simulations. Programs like *bwaves, zeusmp, gemsFDTD, sphinx, etc.* were highly writeback intensive with nearly 30% of the traffic on the main bus being the writeback traffic.

As a result of the speed-up achieved, the group of instructions taking more than 1000 processor cycles was reduced. The results are shown in Figure 18 and Figure 19 for I/O injection rates of 1.2 GB/Sec and 1.8 GB/Sec respectively. Although a majority of the 100 million instructions simulated took only around 100 to 200 cycles to execute due to cache hits, there were instructions which took more than 1000 cycles. Instructions taking more than 1000 cycles refer to those that are affected by the writeback latencies upon a read miss in L2. Hence these additional cycles were mainly due to the memory accesses and main bus contention. More than 90% decrease in the number of such instructions was seen across the benchmark suite and this justifies the speed-ups seen in Figure 17.

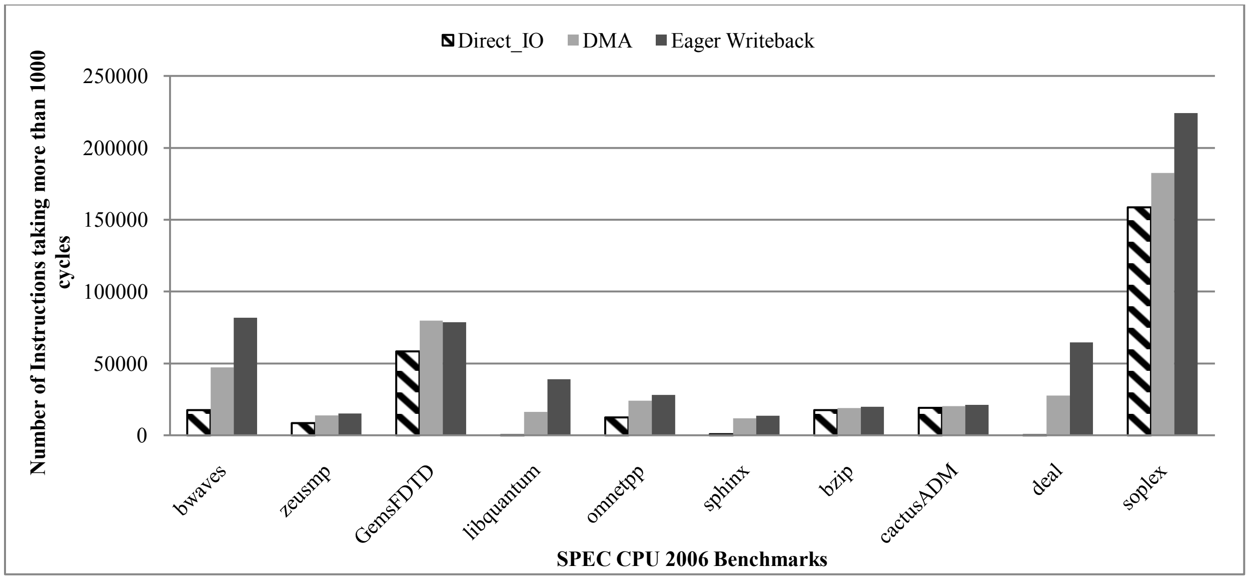

Figure 18: Comparison between I/O techniques and Eager Writeback for I/O rate of 1.2 GB/Sec.

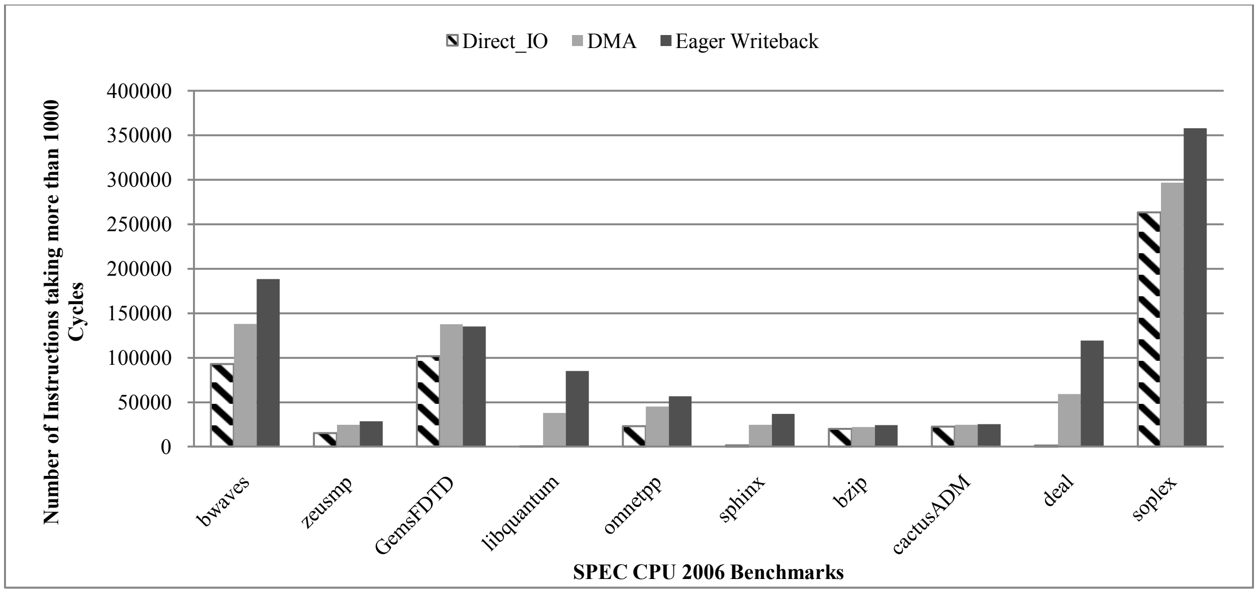

Figure 19: Comparison between I/O techniques and Eager Writeback for I/O rate of 1.8 GB/Sec.

A comparison with the Eager Writeback technique shows that the performance improvement tends to decrease with I/O traffic on the main bus. This is can also be seen in the results from [29].

Secondary bus, because of bus redundancy, scales easily with increased I/O rates. This is best explained with the plot shown in Figure 20 for the GemsFDTD benchmark program. GemsFDTD was chosen mainly because it has a good writeback rate compared to other programs and also shows improvements in CPI in excess of 35% for high I/O rates. With DMA, the results tend to flatten out at large I/O traffic rates. But it is not necessarily true that the CPI keeps improving with increased I/O with the secondary bus. There will be a point where the entire 4.8GB/Sec bandwidth of the main bus would not be sufficient and it may result in queuing delays and CPIs going out of the usual range of 0 to 3 cycles per instruction. That problem is still due to the main bus bandwidth getting exhausted which, happens very rarely as the main bus width is proportional to the number of devices on the bus. Increased number of agents on the bus will surely lead to wider main system bus.

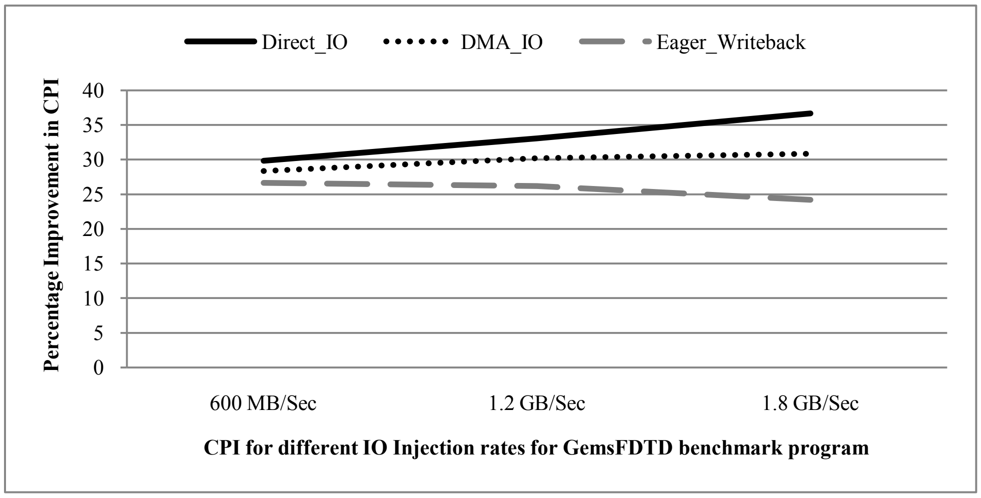

Figure 20: CPI percentage improvement for the GemsFDTD program.

Chapter 7. Future Work

The benefits of the secondary bus were obvious from the simulation results seen in the previous chapter. Though the technique was applied to a uniprocessor system during our analysis, a similar implementation can very much be used even with a multi-core processor. Although the front side bus (FSB) architecture is slowly giving way to technologies such as Quickpath [16] and HyperTransport [17], we still have more room for improvement on the FSB as seen from the secondary bus. Since FSB is a simpler design compared to Quickpath or HyperTransport, secondary bus architecture is worth considering for a dual core or a quad core processor. As seen in Figure 21 and Figure 22 [16] many multi core processors still use FSB as their system bus for communications with the chipset and hence secondary bus can be quite handy in easing congestion. Simulations with the multi core processor environment with the secondary bus can be done similar to the ones in the previous chapter using full system simulators such as GEMS, Simics or the M5.

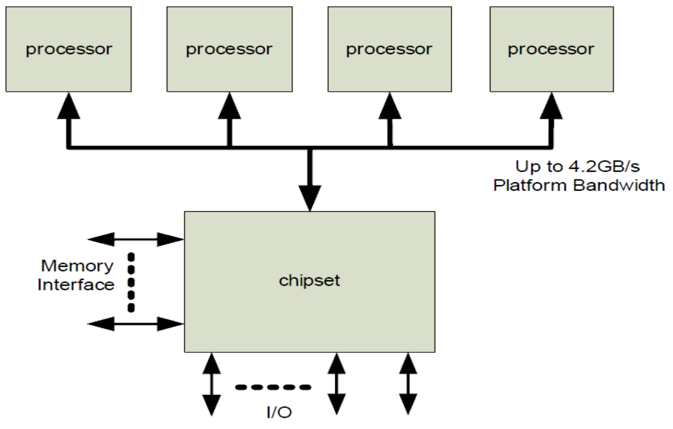

Figure 21: Front side bus architecture in Intel's multi core processors.

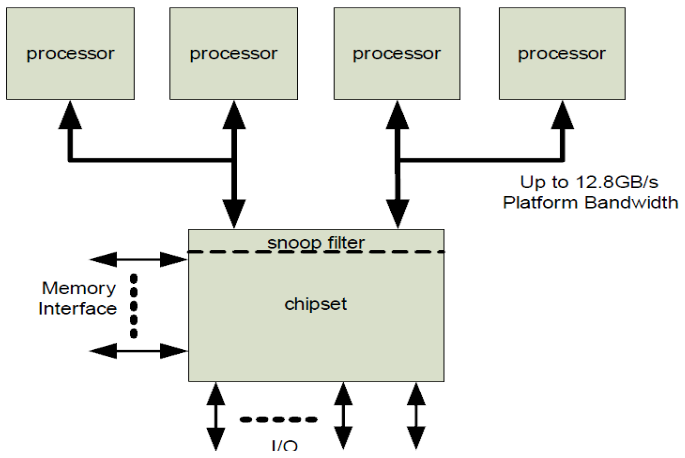

Figure 22: Dedicated FSB for each dual core processor.

Implementation considerations for the secondary bus need to be researched. We have shown that an 8 bit bus with a bandwidth of 600MB/Sec is good enough for the secondary bus, but we should be able to find a technology for realizing the same in hardware. Serial buses such as SATA provide sufficient bandwidths for write back data and can be considered. Wireless link is another technique that can come handy when facing space related constraints. A wireless link however would require the addition of a wireless transmitter and a receiver which may consume slightly more power than a bus based link.

The I/O injection rates were well tested with the proposed architecture and future simulations can use benchmarks that can actually generate the I/O data as well as feed on them. Some web applications are best suited for this purpose. It would call for writing an independent benchmark and then compiling the same for the Alpha benchmark. Modifications have to the made to the bridges and other controller logic to differentiate between the I/O and CPU data when communicating through the memory hierarchy. As mentioned in chapter 5 sim-alpha does not have these extensions built into it.

Chapter 8. Conclusion

A simple solution for reducing latencies due to bus contentions on the main system bus between the CPU and the chipset has been evaluated in this work. The technique of bus redundancy combined with a policy for efficient bus bandwidth management through data traffic distribution have shown to give significant performance improvements compared to existing architectures. Since write traffic on the main system bus is the largest contributor to the queuing delays, separating the write and read data traffic is beneficial. Also, queuing delays were considerably reduced due to the sharing of the bus load by the secondary bus. This improvement was verified across a range of the SPEC CPU 2006 benchmarks which comprised of both CPU and memory intensive workloads to test the architecture.

Performance metrics such as the average queuing delay per instruction and cycles per instruction were used to validate the results and understand the CPU areas that were directly impacted by this architecture. Comparisons were made against two types of I/O techniques namely DMA and Direct I/O and we found that this design would be highly advantageous in the presence of I/O traffic on the main bus. I/O devices and processors can be involved in two types of communications that determine the traffic direction on the main system bus:

1. I/O produce, CPU consume (more than 80% of the I/O traffic are of this type).
2. CPU produce, I/O consume.

We were unable to analyze the second type of traffic as our benchmarks did not generate any I/O traffic themselves. The 'I/O produce and CPU consume' traffic type was simulated and analyzed by creating an I/O device that would pump data at regular intervals either directly the on-chip cache (direct I/O) or to the memory through DMA. In addition to seeing better improvements over the base architecture, the secondary bus approach also proved to be better than Eager writeback for most of the SPEC programs. When larger I/O rates were considered, the gap between Eager writeback and secondary bus widens.

The secondary bus can be implemented in many ways. In our simulations we used an 8-bit wide bus to evaluate the design for different traffic intensities. In the future, various other implementations of the secondary bus can be tried. Split bus or pipelined transactions can be tried on the secondary bus with multiple bit lines. The number of bit lines that can be used for the secondary bus depends on the L2 cache write-back rate. The secondary bus provides excellent benefits for single ported memories at a minimal cost, which consists of a small hardware addition for controlling the bus accesses and a small bus.

Bibliography

[1] G. E. Moore, Cramming More Components onto Integrated Circuits, *Electronics*, Vol. 38, pp. 114-117, 19 Apr. 1965.

[2] Memory Hierarchy [Online]. Available: http://en.wikipedia.org/wiki/Memory_hierarchy

[3] P. P. Chu and R. Gottipati, Write Buffer Design for On-Chip Cache. *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 311-316, Oct. 1994.

[4] T. E. Anderson, H. M. Levy, B. N. Bershad and E. D. Lazowska, The Interaction of Architecture and Operating System Design, *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108-120, 1991.

[5] B. Chen, Memory behavior of an X11 window system, *Proceedings of the USENIX Winter Technical Conference*, Jan. 1994.

[6] D. Nagle, R. Uhlig, T. Mudge, S. Sechrest, Optimal allocation of on-chip memory for multiple-API operating systems, *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 358-369, 18-21 Apr. 1994.

[7] N.P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, 28-31 May. 1990.

[8] N.P. Jouppi, Cache Write Policies and Performance, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 191-201, 28-31 May. 1993.

[9] K. Skadron, D.W. Clark, Design issues and tradeoffs for write buffers, *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp.144-155, 1-5 Feb. 1997.

[10] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A quantitative approach*, Morgan Kauffmann Publishers Inc., 3rd edition, 1996.

[11] *Serial ATA Revision 3.0 specification*, 27 May. 2009, [Online]. Available: http://www.serialata.org/documents/SATA-6Gbs-Fast-Just-Got-Faster.pdf

[12] S. Addagatla, M. Shaw, S. Sinha, P. Chandra, A.S. Varde, M. Grinkrug, Direct Network Prototype Leveraging Light Peak Technology, *Proceedings of the IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, pp. 109-112, 18-20 Aug. 2010.

[13] *PHY Interface for the PCI Express and USB 3.0 Architectures*, 3 Nov. 2009, [Online]. Available: http://download.intel.com/technology/usb/USB_30_PIPE_10_Final_042309.pdf

[14] M.J. Koop, Wei Huang, K. Gopalakrishnan, D.K. Panda, Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand, *Proceedings of the 16th IEEE Symposium on High Performance Interconnects (HOTI)*, pp. 85-92, 26-28 Aug. 2008.

[15] *AGP V3.0 Interface Specification*, Sep. 2002, [Online]. Available: http://download.intel.com/support/motherboards/desktop/sb/agp30.pdf

[16] *An Introduction to Intel Quickpath Interconnect*, Jan. 2009, [Online]. Available: http://www.intel.com/technology/quickpath/introduction.pdf

[17] *HyperTransport I/O Link Specification*, Nov. 2006, [Online]. Available: http://www.hypertransport.org/docs/twgdocs/HTC20051222-0046-0017.pdf

[18] *Career Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, Jun. 2010, [Online]. Available: http://standards.ieee.org/getieee802/download/802.3ba-2010.pdf

[19] D. Tang, Y. Bao, W. Hu and M. Chen, DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance, *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, pp. 1-12, 9-14 Jan. 2010.

[20] R. Huggahalli, R. Iyer and S. Tetrick, Direct cache access for high bandwidth network I/O, *Proceedings of the 32nd International Symposium on Computer Architecture*, pp. 50- 59, 4-8 June 2005.

[21] E.A. Leon, K.B. Ferreira, A.B. Maccabe, Reducing the Impact of the MemoryWall for I/O Using Cache Injection, *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects*, pp. 143-150, 22-24 Aug. 2007.

[22] Z. Chuanjun and F. Vahid, Using a victim buffer in an application-specific memory hierarchy, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, vol.1, pp. 220- 225, 16-20 Feb. 2004.

[23] G. Memik, G. Reinman, W.H. Mangione-Smith, Reducing energy and delay using efficient victim caches," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 262- 265, 25-27 Aug. 2003.

[24] A.J. Smith, Cache memories, *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, Sep. 1982.

[25] J. Tse and A.J. Smith, CPU cache prefetching: Timing evaluation of hardware implementations, *IEEE Transactions on Computers*, vol.47, no.5, pp. 509-526, May. 1998.

[26] D.M. Tullsen, S.J. Eggers, Limitations Of Cache Prefetching On A Bus-based Multiprocessor, *Proceedings of the 20th Annual International Symposium on Computer* Architecture, pp.278-288, 16-19 May. 1993.

[27] The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology, *Intel Technology Journal*, vol. 8, Issue. 1, 18 Feb. 2004.

[28] D. Kroft, Lockup-Free Instruction Fetch/Prefetch Cache Organization, *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 81-87, 12-14 May 1981.

[29] H.H.S. Lee, G.S. Tyson, M.K. Farrens, Eager writeback-a technique for improving bandwidth utilization, *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp.11-21, 2000.

[30] J. O'Farrell and S. Baskiyar, Improved Real-Time Performance Using a Secondary Bus, *Proceedings of the Computers And Their Applications*, Honolulu, HI, March, 2010, ISCA Press.

[31] S. Baskiyar and C. Wang, A secondary channel between cache and memory for decreasing queuing delay, *US Provisional patent application filed no. 61/003,542* on Nov 17, 2007, Auburn University, AL.

[32] *Intel Embedded Pentium processor family Developer's Manual*, December 1998. [Online]. Available: *http://www.intel.com/design/intarch/manuals/273204.htm*

[33] R. Desikan, D. Burger, S.W. Keckler, Measuring experimental error in microprocessor simulation, *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp.266-277, 2001.

[34] T. Austin, E. Larson and D. Ernst. *Simplescalar: An Infrastructure for Computer System Modeling*, IEEE Computer, Volume 35, Number 2, pp. 59-67, Feb. 2002.

[35] R.E. Kessler, E.J. McLellan and D.A. Webb, The Alpha 21264 microprocessor architecture, *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pp. 90-95, 5-7 Oct. 1998.

[36] SPEC CPU2006 Benchmark Descriptions, ACM SIGARCH newsletter, *Computer Architecture News, Volume 34, No. 4*, September 2006.

[37] K. Ganesan, D. Panwar and L. K. John, Generation, Validation and Analysis of SPEC CPU2006 Simulation Points Based on Branch, Memory and TLB Characteristics**,** *Proceedings of the SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking, Section: Modeling and Sampling Techniques*, pp. 121-137, Jan. 2009.

[38] Alabama Supercomputing Authority. http://www.asc.edu/