Multi-core Processors and Caching - A Survey

Jeremy W. Langston and Xubin He
Electrical and Computer Engineering
Tennessee Technological University
{jwlangston21,hexb}@tntech.edu
August 1, 2007

Abstract

Multi-core processors are the industries' current venture into new architectures. This paper explores what brought about this change from a single processor architecture to having multiple processors on a single die and some of the hurdles involved, and the technologies behind it. This is different from past architectures that used multiple, physically separate processors, using multiple sockets. Having each processor, or core, on a single die allows much greater communication speeds between the processors, among other benefits. The biggest pushes for multi-core processors have been the need for multi-threading and multitasking, security and virtualization [1], and physical restraints such as heat generation and die size.

These benefits do not come free. Processor cache, the memory between the main memory and the CPU registers, is the performance bottleneck in most current architectures, and as such, can have vast improvements to the overall system. These caching methods are complex - multi-core processor caches are even more so. This paper will explore some of the research performed on different caching schemes.

1 Introduction

Traditional processor architectures have pushed the transistor count well into the hundreds of millions. These transistors, nano-scale electronic switches, can switch between on and off (1 and 0) states billions of times in a second. Each state and transition requires power. One way to counteract the power consumed is to reduce the size of the transistor. However, the transistor can only shrink so much before the functionality of the electronic switch breaks down and allows current to pass improperly [2]. All of this power consumption leads to heat production, another side-effect of high transistor counts. Yet another side-effect of adding more transistors is the decreasing area on the die for placing them. These issues point toward a shift in architectures: greater parallelism.

Computing has passed the times of batch processing and is well into the era of multitasking. On a single core processor running multiple applications, the operating system acts as a scheduler - switching contexts between the applications. This can require a complete dump of all processor registers and possibly the cache(s), which is costly in terms of completion time. It is obvious that lessening the frequency of context switching will increase the usable cycles of a processor. One way of achieving this is by creating more processors to distribute the load. For example, a computer running two applications will not need to switch contexts if there are two processors working in parallel. This example is simplistic as operating systems often take control, running scheduling and other management tasks in the background.

This parallelism is realized by creating multiple processors, cores, on a single die. Making multi-core processors be effective is not without its challenges however. In order for applications to reap the greatest benefit from multiple cores, the programmer must divide the application into simultaneous threads or be done by the operating

system for multitasking. A thread is a lightweight sub-program that shares the same memory space as other threads under the same program process. This notion of multi-threading is challenging relatively new and isn't yet taught to be as fundamental as, say, data structures. There is also an architectural design challenge for multi-core processors: the caching scheme to be used.

The remainder of this paper is divided as follows: section 2 gives a brief background into multi-core architectures and cache techniques; section 3 depicts how multi-core processors can and are used; section 4 tells how to critically analyze the designs before they are fabricated, and after; section 5 states some of the previous and current work being done.

2 Background

2.1 Computer Architectures

Past architectures have included multiple physically separate processors. Those architectures fall far behind the multiple on-chip processors due mainly to wire delay and caching techniques. Wire delay is the time it takes for data to traverse the physical wires. This can have a drastic effect on frequencies. As such, structures requiring high throughput between each other are placed in close proximity. There is also the added problem of limited intra-processor communication pins for multiple separate processors - a problem not seen in multi-core processors.

2.2 Cache

Computer cache plays an intermediary role between main memory and the processor. The objective is to lessen the number of accesses to main memory, which are relatively slow due to the memory type it is (e.g. double data rate synchronous dynamic random access memory, or DDR SDRAM). Cache is made from static RAM (SRAM), built from flip-flops, to provide faster access times. DDR SDRAM is slower, but cheaper. SRAM is a up to four times larger than an equivalent DDR SDRAM module. Since cache is typically found on-die with the processor, area is at a premium and this decides the amount to be included.

Cache is about having memory stored locally for items that will be used in the near future. To aid in finding this memory, designers begin with locality of reference [3]. This is the idea that memory located near previously used memory will likely be accessed. The term "near" can be adjectified three different ways: spatially (physical nearness), sequentially (physically right after another), or temporally (memory reused in the near future). This only depicts what memory would be used. In order for cache to be effective, there are several issues to be dealt with: initial placement, identification, replacement, and write strategies [3]. These have to deal with the fundamental cache element, a block. Changing the block size, as well as various other changes such as mapping, change the pertinent cache aspects: cache hit and miss rates, miss penalties, and time to hit [4].

A block is typically around 4 to 32 kilobytes, but the size is up to the designer. Increasing the block size will decrease the amount of cache misses as more data and instructions are in each block. However, cache schemes are a give and take procedure. While a bigger block size decreases the miss rate, the miss penalty goes up. This miss penalty is the time it takes to get a new block from main memory into the cache, and replace another block. The simplest way to counteract this miss penalty is to increase the amount of cache memory. This is a commonly used optimization technique, but can only be done at the cost of hardware complexity and thus more area on the die is consumed, more power consumed, and more heat generated. The third easiest technique is done by adding more levels of cache. This works in the same way as main memory does for hard drives and CPU registers do for main memory. An Intel Pentium 4 processor uses two cache levels. Level 1, referred to as L1, is 8kB and 16kB, while level 2, L2, is 1MB. The sizes have continually been pushed and, at the time of this writing, an L2 size of 4MB is not uncommon. It is also quite common to have two L1 caches per processor/core. This separates the data from the instructions. The L2 however is made up of both

data and instructions; hence this L2 arrangement is referred to as unified.

Other optimization techniques can be performed, but more information is needed about cache architecture. Some techniques are straightforward while others are very complex. One of the primary aspects of caches is the type of mapping strategy: direct, fully-associative, and set-associative [3]. These depict how the blocks are stored and retrieved. The CPU will make requests of main memory for a particular address, which goes through the cache. The cache must translate this main memory address into a block location within the cache. Without delving into the exact details, the addresses are broken up into 2 or 3 different fields, depending on the mapping strategy [5]. When the data/instructions are copied from main memory to the cache, these fields determine where they are stored. The simplest strategy is direct. Each block in main memory has exactly one and only one location in cache it can be copied to. See figure 1 for an example. This strategy is less costly as no searching is required. However, if thrashing occurs, when one cache block is continually swapped between two or more memory blocks, the overhead becomes an issue.

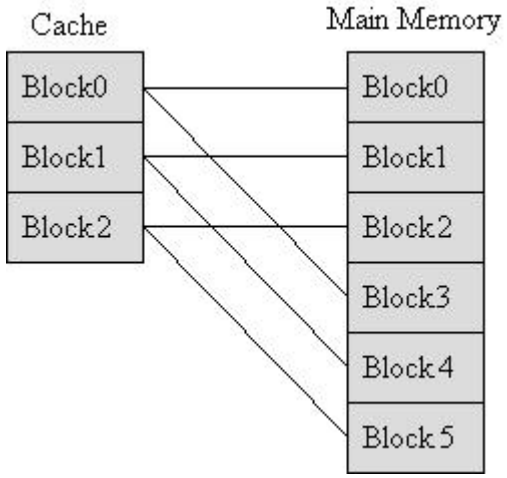

Figure 1: Direct mapped cache, from L. Null

Fully-associative mapping is the opposite of direct mapping in that the memory blocks can be stored anywhere in the cache. In this way, the entire cache must be searched for each memory access. This requires more hardware and is thus very costly. A combination of the two extremes, direct and fully-associative, forms the most common mapping strategy: set-associative. Here, the cache is broken up into separate sets. Each set is made up of two or more blocks. A two block set-associative mapping is referred to as 2-way, because the data retrieved from main memory can be put in two different locations, instead of just one. This allows the cache some flexibility and limits the amount of thrashing that could occur.

Writing to cache from the CPU presents another opportunity for optimization. There are two simple write policies: write-through and write-back [4][5]. During a typical write, the CPU stores its computed data to a location in cache, which is stored back into main memory. These two policies differ in when they store the updated cache contents to memory. Write-through stores the data into the cache and then into the main memory. Write-back stores the data in the cache, and only writes to main memory when evicted. A write to memory is even slower than an access. Procrastinating the memory write until eviction can minimize the number of memory write procedures. A more advanced write optimization involves buffering the data to allow memory reads to preceed the writes, as they are faster.

Technique	Hit Time	Miss Penalty	Miss Rate	Complexity
Larger block size		-	+	0
Larger cache size	-		+	1
Higher associativity	-		+	1
Multilevel caches		+		2
Read priority over writes		+		1
Avoid address translation during cache indexing	+			1

Table 1: Simple optimization techniques. From Hennessy, Patterson.

The preceeding table presents some of the mentioned optimizations, as well as some others.

Cache optimization is a widely researched topic and the different schemes are endless.

3 Uses

3.1 Servers

Servers have a direct application for multi-core processors. A server can potentially have many simultaneous connections to many users. To accept these connections, the server will either spawn a new process or fork off a new thread. This allows the main process/thread to continue to wait for connections. The operating system can then allocate these workloads across the available cores. It is becoming common to have four or more cores for server applications. This works well with long running connections.

3.2 Consumers

The consumer market has adopted these new processors, banking on the multi-tasking parallelism granted by the multiple cores. Since the time of Windows and it's multi-tasking ability, this concept has become a mainstay. It it not uncommon to be actively running 5 or more programs, with another 50 running in the background. These applications reap direct benefit from a multi-core architecture by either multi-threaded programs or via scheduling by the operating system.

Multi-core processors are not limited to traditional computers. Two such examples are the Cell processor [6][7] and NVIDIA Tesla GPU [8]. Both of these are used for graphics rendering, a very processor intensive task. The Cell processor, in use by the Sony Playstation 3, utilizes 8 heterogeneous cores. The Tesla GPU has 128 cores and is used for high performance computing.

3.3 Virtualization

The idea of virtualization is nothing new. It tracks back to the days of mainframes. At the time, having many computers could not be justified either because of cost or under-usage. Now the costs are far lower. However, one thing remains to be true: under-utilization. A system administrator can configure the computer to "virtualize" its devices, or operating system, to allow one ore more simultaneous virtual machine(s) to use the computer as if each virtual machine (VM) was its own computer. Doing so allows the computer to be further utilized, instead of constantly spinning in an idle loop. There are more uses than just these for using VMs, including server consolidation, IT center area restrictions, dynamic optimization, security, and hardware virtualization for multiple parallel-running operating systems [15].

4 Analysis Techniques

In the theoretical design of an architecture, one uses mathematical equations to verify the performance. This is very prevalent in cache design. Miss rates are a common metric of cache implementations; where miss rate is the ratio of misses to memory accesses. This simple analysis is augmented by involving the times associated with miss penalties and hit times. From [4], the average memory access time (AMAT) in seconds or clock cycles can be found by

$$AMAT = Hit\ time + (Miss\ rate * Miss\ penalty)$$

where hit time is the time it takes to get a memory location and miss penalty is the time involved when the requested memory is not found in the cache. Miss penalties are much larger than hit times, as the cache must repopulate a block with the corresponding data/instructions located in main memory. Other equations involve concepts such as out-of-order processing, multi-level caches, etc.

The most common way to test configurations before a complete physical implementation is via emulation and simulation software. Basic structures are tested for functional and timing requirements by giving a series of test cases to simulation software. This simulator will run the cases through the compiled logic (derived from an HDL at the hardware level). In [9], hardware prototyping and testing is analyzed using a Xilinx Virtex-II Pro FPGA. Using an FPGA as a test bed gives great reconfigurability. Due to the complexity of even a simple processor architecture, these methods cannot be done satisfactorily as a whole. As stated in [10], random program generators and simulation methods are used to test the basic structures when combined. Lewin goes on

to introduce automatic architectural test program generators to verify proper working conditions of complex systems, such as multi-core processors.

Upon implementation, benchmarking software, such as SPEC CPU2006 [11], is used to test the many aspects of a processor. In the CPU2006 package, 29 different benchmarking programs test all areas of the processor using practical applications. The results of the testing are compared against preset standards.

5 Previous Work

Industry giants Intel and AMD started shipping their multi-core processors during 2006 to the user and server markets. The AMD Athlon 64 FX dual-core processor has two L1 caches, data and instruction, and one L2 cache, unified, for each core [12] (see Figure 2). Intel uses a shared L2 cache in what is referred to as the "Advanced Smart Cache" [13] (see Figure 3). This implementation dyamically shares its second level cache to utilize 100% of the available cache, thus reducing the cache misses and increasing the performance. For a further breakdown of the differences between these processors, see table 2.

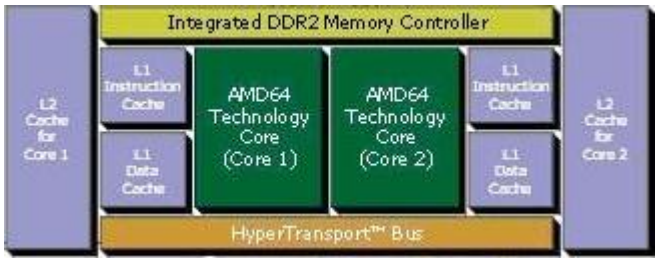

Figure 2: AMD Athlon 64 FX Architecture. Image from AMD.

A similar concept [14] proposes is a non-uniform cache architecture to share cache between cores dynamically. This architecture addresses the cache pollution that occurs when one core uses cache space unnecessarily and intrudes on another core's space. The proposal is done with a quad-core processor and three levels of cache. The third level, L3, is partly shared and partly private. Each core is allotted a certain amount of space in L3 to be private and cannot be intruded upon. The remaining cache is shared between all four cores. There are three different events that occur in the cache: a hit occurs in the private L3 - a normal hit; a hit occurs in the shared L3 - missed in private, found in shared, moved to private; and a cache miss - inserted into the private cache from main memory. The proposed Sharing Engine determines the best cache allocation and partitioning, the sharing of cache space, and the replacement policy. Naturally there is an inherent cost associated with the increased complexity of such an architecture.

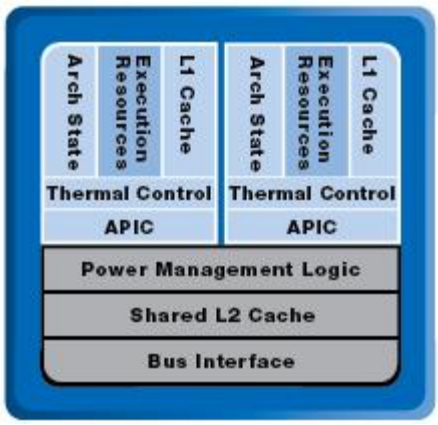

Figure 3: Intel Core Duo Architecture. Image from Intel.

Other more exotic research has been done involving virtual machines on multi-core processors. In [15], the idea of specializing the cores for virtual machines. The heterogeneity can be observed from subtle differences like sizes of cache, or bigger differences such as instruction sets and operating frequencies. They proposed two main designs: a single virtual machine core shared by all other general-purpose and specialized cores (for system virtualization); or each general purpose core can have a virtual-machine-specific core (for process virtualization). The concern with this architecture was with the context switching overhead from swapping traces.

System security and dependability is addressed in [16] with an "integrated framework for dependable and revivable architectures", or INDRA. The application is for recovery of vital network ser-

Processor	CPU Speed	# of Cores	L1	L2	Technologies
Intel Core 2 Duo E6850	Up to 3GHz	2	Data & Inst. for each core: 32kB, private, 8-way	4MB, unified, shared, 16-way	Advanced Smart Cache
Intel Core 2 Duo E4500	Up to 2.2GHz	2	Data & Inst. for each core: 32kB, private, 8-way	2MB, unified, shared, 8-way	Advanced Smart Cache
AMD Althlon 64 X2 and Opteron	Up to 3GHz	2	Data & Inst. for each core: 64kB, private, 2-way	2MB, unified, private, 16-way	

Table 2: Features of some multi-core processors and their caches. Data collected from Intel and AMD datasheets.

vices from remote exploit attacks. INDRA uses a core set at a higher privilege that is protected from remote attacks, a resurrector, and monitors the execution of the other cores, the resurrectees. To further shield the resurrector from attacks, measures such as using different operating systems or changes in the BIOS. System recovery is enacted after the resurrector discovers an attack; then the resurrector stops the resurrectee, recovers its old state, and stops the damage done by the attack. They note three metrics that judge the performance of the system: remote exploit attack immunity, detectability, and the overhead induced. Multi-core processors are used due to the high amount of intra-core communication needed for transferring state information.

6 Summary and Conclusions

Multi-core processors are already expanding their niche and are finding many new and creative uses. Due to physical limitations and increased multi-tasking requirements, the multi-core architecture is expected to become the standard over the single-core predecessors. Parallel programming and operating system collaboration remain key in the proper fulfillment of a multi-core processor's usefulness. Further caching schemes, both specialized and general, will continue to be honed, narrowing the performance gap between the processor and main memory. This new area in computing is exciting and possibly the most challenging yet.

References

[1] Advanced Micro Devices, Inc., "Multi-core Processors - The Next Evolution in Computing," White paper, 2005.

[2] D. Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *Computer.org*, IEEE, pp. 11-13, May 2005.

[3] V. P. Heuring and H. F. Jordan, *Computer Systems Design and Architecture,* Prentice Hall, 2nd Edition, 2003.

[4] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, 4th Edition, 2007.

[5] L. Null, J. Lobur, *Computer Organization and Architecture,* Jones and Bartlett Publishers, 2003.

[6] M. Gschwind, "The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor," IBM Research Division, 2006.

[7] IBM Research, http://www.research.ibm.com/cell/.

[8] NVIDIA Corporation, http://www.nvidia.com/object/tesla_gpu_processor.html, 2007.

[9] C. R. Clark, R. Nathuji, H. S. Lee, "Using an FPGA as a Prototyping Platform for Multi-core Processor Applications", Georgia Institute of Technology, Atlanta, GA.

[10] D. Lewin, D. Lorenz, S. Ur, "A Methodology for Processor Implementation Verification", Technion, Haifa, Israel.

[11] J. L. Henning, SPEC CPU Subcommittee, "SPEC CPU2006 Benchmark Descriptions", Standard Performance Evaluation Corporation, 2006.

[12] Advanced Micro Devices, Inc., "AMD Athlon 64 FX Processor Key Architectural Features", http://www.amd.com/us-en/Processors/ProductInformation/.

[13] O. Wechsler, "Inside Intel Core Microarchitecture", Intel Corporation, White paper, 2006.

[14] H. Dybdahl, P. Stenstrom, "An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors", HiPEAC Network of Excellence.

[15] D. Upton, K. Hazelwood, "Heterogeneous Chip Multiprocessor Design for Virtual Machines", University of Virginia.

[16] W. Shi, H. S. Lee, L. Falk, M. Ghosh, "An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors", Georgia Institute of Technology, Atlanta, GA, 2006.

[17] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual", 2007.

[18] Intel Corporation, "Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequences", 2007.

[19] Advanced Micro Devices, Inc., "AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet", 2007.