Multithreaded and Multicore Architectures

By Luke Gebauer, Alex Penman, Luke Plewa, Sanat Sahasrabudhe

Table of Contents

1. Abstract

This research paper takes a look at the current state of multithreaded and multicore architecture and designs in today's computing world. It first looks at current models of multithreaded architecture, exploring the different methods and challenges that face it. Different models of multicore architecture are then presented along with some of their performance ratings and design options. After discussing some of the attributes and specifics of multicore designs, an argument for the need for further advances in uniprocessors for the benefit of multicore designs facing the challenges of Amdahl's Law is presented. Lastly, this paper examines the impacts of designing multithreaded platforms specific to a software application's needs and bringing software closer to the hardware. Ultimately, the purpose of this paper is to present some of the modern designs and practices that are being used in today's computer processing and determine the effect that such architectures such as multithreaded and multicore have had.

2. Multithreaded Architecture Models

One of the main ways to implement multithreading into computer architecture is through block multithreading. Also called cooperative multithreading, this model switches over to a new, ready thread whenever it encounters some sort of stall in the current running thread. A common stall would be a cache-miss, as data must be pulled from memory in order to continue execution. However, by switching over to another thread, this flow of execution can follow through with a smaller margin of stoppage time. Once the stall is complete and the original thread is ready to run again, it switches back to run the original thread. This model is similar to how interrupts are handled. When an interrupt occurs, a context switch takes place and a different function is carried out. Ideally, the time to switch between threads would need to be as small as possible in order to make this most efficient. Additional hardware is required to include separate register sets for each of the potentially running threads. By doing this, the threads run virtually independent of each other and a switch only takes one clock cycle to complete. If the extra hardware was not involved, the saved register values would need to be pulled from memory and the current registers saved for the switch back. In the case of a simple cache-miss in the original thread, this longer switch could potentially take longer than it would to just wait for the thread to come back from the stall.

Another mode is interleaved multithreading or barrel processing. Similar to block multithreading, the CPU swaps between different threads, only this time a swap occurs on every clock cycle rather than only when a stall is hit. The idea behind barrel processing is to cut down

the data dependency between instructions so the flow of execution can run smoothly even without feed forwarding lines. A nice thing about this model is that you can guarantee that a thread will complete in a precise time regardless of other threads running into lock ups or stalls (assuming the thread itself does not encounter any of these). This is because, the switch between threads happens every cycle no matter the state of the threads so the execution continues at a current rate. Since this process is similar to block threading in that it switches back and forth, separate register sets and PC's are required for each of the different threads.

Simultaneous multithreading differs from the previous models in that it allows for multiple instructions to be executed in a single pipelining stage which can be referred to as thread level parallelism. The amount of simultaneous threads running on a given chip is up to the designers but because of physical restrictions, this number is usually limited to two. Through this design, multiple threads run at the same time as long as the instructions do not rely on the same hardware. This allows for an increased throughput with each clock cycle, not just the entire instruction cycle. Just as with the other multithreading techniques mentioned earlier, this requires some extra hardware. The register set must be increased to accommodate simultaneously running threads. Even though more is needed, it doesn't need a completely separate register set like before, the data from the threads can be stored in any register as long as they don't get mixed up by the user. There is also a need to be able to fetch instructions from multiple threads at the same time in a way that doesn't slow down efficiency.

3. Multicore Architecture Models

Multithreading is enhanced by using multiple components. The more components you have, the more places you have to spread the workload across. This is the basic concept behind using *multicore microprocessors*. Having more processors, which adds more components, means we can make multithreading more powerful.

Multiple processors exist in two forms: on a single chip, which is the relatively new multicore design, and between multiple chips on a supercomputer or server, which is the older design. *Single chip multicore* has become more attractive because it has become more affordable due to advancements in technology. Patterson and Hennessy say, "The number of cores that can fit on a chip is expected to double every two years." This has led to allowing even the cheapest computers to possess multiprocessors. Instead of having a complex uniprocessor, focused on improvements in clock cycle and CPI, the focus on improvement for multicore is adding more cores to the chip. A multicore chip has increased performance per watt and by space thanks to its simple microarchitecture compared against a uniprocessor of the same computing power.

As the number of cores increases, it will become more helpful in being able to measure multicore performance. Floating-Point Operations (FLOPs) are based off of operations using the popular data type floating-point, and are a good unit to base performance off of. Performance is modeled through a *roofline diagram*, where attainable GFLOPs/sec, the unit of performance, are measured along the y-axis and FLOPS/byte ratio, or the arithmetic intensity, increases along the x-axis, as seen in Fig. 3.1.

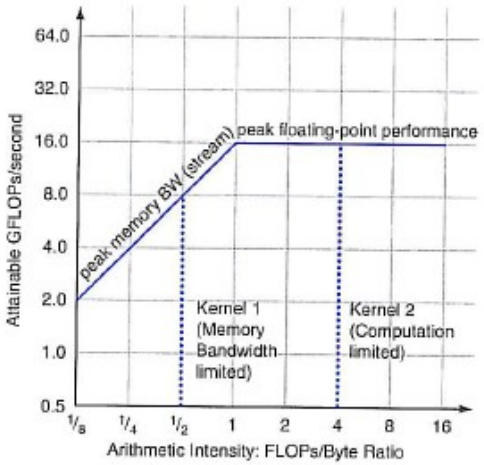

Figure 3.1: Roofline Diagram

Attainable GFLOPS/sec = Min (Peak Memory Bandwidth X Arithmetic Intensity,
Peak Floating-Point Performance)

The Peak Floating-Point Performance is the limitation of the computing power, which is a hardware limitation and cannot be surpassed. Up until that point, the memory bandwidth limitation determines the performance based on arithmetic intensity. Arithmetic intensity is measured in FLOPs/byte, and is calculated by taking the total number of floating-point operations of a software program divided by the total number of data bytes accessed by that software program's execution from main memory. GFLOPs/sec multiplied by FLOPs/byte gives the bytes/sec bandwidth ratio which determines the slope of the peak memory bandwidth line.

A design decision in multicore processor architecture is whether to use *identical cores* or *asymmetrical cores*. The varying cores allow the cores to be specialized to specific tasks, but identical cores may be simpler to design. For instance, a computer can possess a simple core that handles easy threads and a complex core that handles hard threads. This way, the computer has to make a choice in which core is better suited to handle a specific task.

Choosing between identical and asymmetrical cores makes the performance of applications difficult to predict. This unpredictability can be an undesirable effect for a software developer as certain applications have a chance to lose performance. In order to avoid this loss in performance, software must be developed so that it can adjust to the variances in computational power between cores. This can be done through the operating system kernel, by scheduling tasks in a way which predicts these asymmetrical cores. Another way to reduce this side effect is to give the complex core less responsibility of overall computing. This way, a software application can assume it is running on the simple core and that assumption will be less likely to be wrong. With these measures taken, an asymmetrical multicore processor is a beneficial multicore architecture design because of the diversity in computing power.

4. Uniprocessors

Although the move towards mulithread, multi-core processing in today's computing standard may seem to suggest to some computer architects that the solution to increasing processing speeds and efficiency will simply be a matter of adding more identical cores to a processing unit and then splitting up the application to be run into more threads, there are some computer architects and scientists who believe that advances in uniprocessors (optimizing the processor's single-threaded abilities) will still be required for further advances in computer

processing with asymmetrical core architecture. This controversy between simply tying together more identical, latest-generation processing cores to increase overall processing power and speed or to accomplish this end by increasing the power and speed of the individual processors has been going on in the computer architecture world ever since Intel introduced its first 4004 microprocessor. At the time, many computer architects and scientists thought that the world's greatest computer could then be built by simply stringing together a bunch of these 4004's, but Intel's legacy of microprocessors since that model has made it obvious that improving the power and speed of uniprocessors was still a key component in increasing overall computing performance.

One of the reasons that proponents of further advances in uniprocessor speeds and power believe that this is still an important area of research and engineering is because it will increase the processing times of pre-existing or legacy programmers that were written for single threaded architectures. Although having more cores available for processing may seem to offer more processing power, this will only be true insofar as the programs that they are running are able to divide themselves into separate threads for these processors to handle. For older, legacy applications that were written for single-threaded architecture, having a number of extra cores that sit there in an idle state while the program is ran on a single one will do nothing to increase the program's processing time.

The other main reason that proponents of uniprocessor advancement for an asymmetrical core configuration think this is still a necessary area of research and design in order to further increase computer performance is the boundaries that are placed on computer processing by Amdahl's law. Amdahl's law essentially states that the increase of speed of a program using multi-core processing in parallel computing will be limited by the sequential part of the program being run. In essence its saying that regardless of how many processor you put on a chip, that chip will only be able to run a program as fast as it can run the slowest sequential part of that program. It's the old bottleneck problem. This is why many computer architects and scientist believe that exploring further advances in uniprocessors is still an important area of research. As Yale N. Patt said in the 2007 Workshop on Computer Architecture Research Directions, "I'm saying that you need one or more fast uniprocessors on the chip. Without that, Amdahl's law gets in the way. That is, yes, your Niagara thing [referring to a processor configuration of many lightweight processors tied together] will run the part of the problem that is embarrassingly parallel. But what about the thing that really needs a serious branch predictor or maybe a trace cache?". Therefore, the common, middle-ground solution to this issue of multiprocessor versus uniprocessor advancement is to build a type of asymmetrical processor architecture in which a number of lighter-weight processors effective for parallel processing are coupled with a few high-power, heavy-duty processors that take care of the more robust and serious computational serial parts of a program.

5. Multithread Specific to Software

Multicore CPUs run applications that are usually multithreaded, so it is recommended to have software which can make this efficient. One method of doing this is to use the system known as the message passing interface (MPI). MPI is communications protocol that was developed in the late 1980s and early 1990s to be used in parallel computing applications, although it can also be used for point-to-point communication. A standard for MPI was developed, which defines syntax and semantics for library functions useful for portable message passing programs in C and Fortran.

Message passing can be seen as a form of interprocess communication, including when multiple processes are running at the same time. It can be distinguished from calling functions in that it uses significantly more memory for copying the current argument into a new message, longer transfer time due to asynchronous passing, and the state of the message handler remaining in the same state regardless of the behavior of the sender or client process; in fact, it is a form of abstraction that hides state changes that may be used for implementation of sending messages. Message passing can be either synchronous or asynchronous. Synchronous passing requires the sender and receiver to wait for each other, but no buffering is involved and the sender will not continue until the receiver is ready so data can be stored by the receiver. Asynchronous passing allows the sender and receiver to overlap in their communication; however, deadlock is a major problem in this type of communication. Deadlock is common in multithreading, and occurs when more than one process tries to use the same resource at once, which can cause messages to be dropped and thus disabling communication. A common solution to avoiding deadlocks is to use locks, or mutual exclusion (mutex). In this method, any process that tries to use a resource must be assigned a lock. Once a process acquires this lock, other processes cannot access the resource until the current process is done using it. The next process in queue for the resource can now acquire the lock. In message passing, the message handler controls access to the resource by processes. An example of message passing in use is when a client is communicating with a web server.

MPI includes the following basic concepts: communicator (connecting groups of processes in the MPI session), point-to-point communication, collective functions, and derived datatypes (such as MPI_INT and MPI_CHAR). Point-to-point and collective functions are the preferred ones for multithreading. Point-to-point normally describes a single process sending a message to another process, but this can also be done by among multiple processes at once, provided that proper resource-sharing controls are in place. MPI has specific mechanisms for handling both blocking and non-blocking communications; in a blocking communication, a subroutine does not return until it is completed or stopped by an exception. Blocking can also be enforced through locks. Collective functions deal with communication among all processes in a group, among all nodes. For example, MPI_Bcast sends a message from one node to all others within the group; such communication is common in networks, where it is necessary to have multiple processes to maximize bandwidth.

OpenCL (Open Computing Language) is another framework designed for parallel computing. OpenCl runs across heterogeneous platforms made up of CPUs, Graphics Processing Units (GPUs), and application programming interfaces (API). A common use of OpenCL is to allow an application access to a GPU for multiple non-graphics processes. OpenCL has often been compared with Compute Unified Data Architecture (CUDA), another parallel computing platform. Both support graphics and API, but OpenCL is more portable. CUDA is directly interfaced to the platform it executes. Currently, there are efforts to run OpenCL for multithreaded applications using field-programmable gate arrays (FPGA). The goal here is to use it for application-specific processors on different types of hardware.

6. Conclusion

Multithread and multicore architectures show progress in the power of computing. They are innovations upon standard, singular designs. There is room to grow for both types of architecture, but there's no telling when the next, great computing idea will come up. Multithreading through the block and interleaved designs is the current architecture, but the

needs of software could drive the ideas for new designs. Multicore design can be implemented either through identical cores or asymmetrical cores. There also advances in uniprocessors for the benefit of multicore designs facing the challenges of Amdahl's Law. The advances in technology presented in this research show that there are new ways to improve already existing computing design. These are all ideas which improve upon existing models and are plenty useful.

7. References:

Multithread Computer Architecture: A Summary of the State of the Art
By Robert A. Iannucci
http://books.google.com/books?id=IkdqkJq2h2kC&lpg=PR13&ots=UNe_P08bDb&dq=multithreaded%20computer%20architecture&lr&pg=PA1#v=onepage&q=multithreaded%20computer%20architecture&f=false

Multicore and Multithreaded Processors Lecture
By Daniel J. Sorin, 2009 @ Duke University
http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2009/lectures/8.2-multicore.pdf

The Impact of Performance Asymmetry in Emerging Multicore Architectures
By Computer Sciences Department University of Wisconsin-Madison
http://www.cs.washington.edu/education/courses/cse590g/05au/08A-03.PDF

Design methods of multithreaded architectures for multicore microcontrollers
http://ieeexplore.ieee.org.ezproxy.lib.calpoly.edu/xpl/articleDetails.jsp?tp=&arnumber=5873041&contentType=Conference+Publications&matchBoolean%3Dtrue%26searchField%3DSearch_All%26queryText%3D%28%28benefits+of+multiclustered+OR+multithreaded+OR+multicore%29+AND+processor+architecture%29

Multi-core Architectures
By Jernej Barbic
http://www.cs.cmu.edu/~fp/courses/15213-s07/lectures/27-multicore.pdf

"NVIDIA Developer Zone." *OpenCL*. N.p., n.d. Web. https://developer.nvidia.com/opencl

"Message Passing Interface (MPI)." *Message Passing Interface (MPI)*. Lawrence Livermore Laboratory, n.d.
By Blaise Barney
https://computing.llnl.gov/tutorials/mpi/.

Computer Organization and Design, 5th Ed.
By D.A. Patterson and J. L. Hennessy

8. Work Breakdown

The following sections apply to both the presentation and the paper report materials. Each team member was given 1/4th of the bulk of the presentation and report divided equally.

- Luke Gebauer: Abstract (report), Uniprocessor
- Alex Penman: Introduction (presentation), Multithread
- Luke Plewa: Conclusion, Multicore
- Sanat Sahasrabudhe: Software