Performance Issues on Multi-Core Processors

Ransford Hyman Jr.

***Abstract*—Multi Core Architectures has been the new generation for processors of today. With Moore's law constantly growing and the frequency reaching a maximum for single CPU chip designs, manufacturers have transformed their designs to multi-cores, where each core is much smaller and has less functionality then a CPU. These cores work together in order to process a job in an effiecient manner. The use of multiple cores allows for the frequency of the processor to be reduced, thus reducing the temperature of the system. With multi cores, instructions are allowed to run on individual cores simultaneously, which increases the amount of parallelism. Today we see processors being built with as many as 8-cores on a single chip. This number is subject to increase, but we must face the issues that the smaller core designs present today. In this paper, we focus on the challenges and the factors that are being researched to increase performance. We look at Thermal constraints and how overheating effects a multi-core processor. We investigate various techniques used for the design of the interconnection network. We examine memory issues and how it effects the performance, in relation to latency. Finally we look at parallelism and the challenges it brings in today's multi core processors.**

***Index Terms*—Multi core processors, performance**

I. Introduction

In today's technology, the demand for higher performance computing (*HPC*)increases rapidly. We are reaching a time where processors executing in gigaflops will be outdated. As Moore's Law continues to grow, we see a shrink in transistor sizes, therefore allowing us to pack more on a single die. But what happens as this trend continues to grow? On the positive side, we have the ability to increase throughput, which is always the goal, by increasing the number of cores on a chip. With more cores on a chip, we can assign special tasks to certain cores creating a *Network on a chip* (NoC). With these performance gains comes performance issues. There are overheating problems, reliability of the chip, algorithm design challenges, and scalability(amount of performance proportional to the number of cores [10]).

Performance is the motivating reason behind computing. Scientists want to compute large and difficult mathematical equations in an expeditious fashion. Video game enthusiasts want real time environments and realistic graphics. Servers need to process large amounts of data, for many users at one time. All of these applications need high performance from the CPU. Performance can be viewed as

$$Performance = IPC \times CPI \times ClockCycle \quad (1)$$

where *IPC* is the amount of instructions per cycle, and *CPI* is the number cycles per instruction. The IPC has been increased using *instruction level parallelism* (ILP) and *thread-level parallelism* (TLP). The CPI has been reduced through the use of *pipeline techniques*. Increasing the *Clock cycle* was once a solution to performance enhancement but as technology progressed, we later realized that this was not the best idea. Fast clock cycles has the ability to create one of CPU's worst nightmare, high power consumption. Power can be defined as

$$P = \frac{E}{T} = \alpha V_{dd}^2 C_{load} f_{sw} \quad (2)$$

where α is the activity factor (*which equals 0.5 in dynamic power*),V_{dd} is the supply voltage, C_{load} is the load capacitance and f_{sw} is the switching frequency.

The first Dual Core Processor was introduced by Intel in 2005. Before 2005, multiple single core chips were synchronized together to enhance performance. In 1996, Intel built a supercomputer holding 10,000 Pentium pro chips which consumed 500 KW of power and 500 KW of cooling [11]. High power consumption causes the chip to give off large amounts of heat (*heat dissipation*) and makes the chip unreliable in its calculations, thus large amounts of cooling is required. On a lower level, Intel's Pentium 4 processor had reached its clock frequency limit at around 4Ghz due to overheating issues [12]. Processor designers had to look into a way to tackle this issue, while maintaining an increase in performance. This is what brought about the dual core era. Dual core processors have the ability to complete more operations in the CPU at a lower clock frequency. The concept behind multi-core is to use multiple, but simpler CPUs (*Cores*) to complete multiple instructions simultataneously. This allows an increase in ILP because more instructions can be processed throughout the cores, and with each core having mulithreaded capability, we can also increase the amount of TLP as well. Today's processors have dual, quad and up to eight cores on a single chip. There are multi-core designs which uses homogenous cores(*complex cores that perform the same operations*) and/or heterogenous cores(*cores that are delegated to different tasks*)Most of the multi-core designs allow for one or more of the cores that are non-operational to turn off for power savings [12]. Even with this abundance of hardware, the *scalability* of

the multi-core designs have not reached its full potential. Lack of *Parallelism* is a contributing factor to why we are unable to achieve this goal. Parallelism gives us the opportunity to increase throughput of data, while maintaining a low power consumption. Research in both hardware and software are looking for ways to overcome this challenge. At Intel, software develepment researchers are faced with the challenge to create applications and algorithms that exploits the multi-core design effieciently and avoiding deadlocks and race conditions among threads. There has also been research to obtain *parallelism* on the compiler level, but this would not be as beneficial as parallel programming. [10]

The future of multi-core is near, but as we expand, the importance of these issues increase. In this paper we want to discuss these issues and how the research community are trying to solve these problems today. We do this by focusing on:

1) **Performance enhancements using the multi core design.**
2) **Thermal issues in multi core and its correlation with performance.**
3) **Interconnection network and its affects on performance.**
4) **Effects of parallelism in correlation with performance.**
5) **Cache and memory issues and their relationship to performance in multi-core processors.**
6) **A future outlook on what is to come in the multi core era.**

In performance enhancements, we want to define performance, and discuss various ways to maximize performance in multi-core design. In thermal issues, we discuss the fallacy of heating in multi-core designs and ways to reduce it. With Interconnection networks, we want to explain how different network techniques relate to performance, and new ideas in networks, such as *NoCs*. With parallelism, we focus on the hardware and software challenges to achieve this within multi-core processors. We talk about the cache/memory issues in relation to performance of multi-core designs and finally we give a future outlook on multi-core processors. This paper should illustrate examples on each topic to convey the information in the best manner possible.

Of course there other equally important issues, such as *reliability*,*energy efficiency*, and even *area optimization* as we move into the nano-scale arena, but in this paper we want to focus specifically on performance. If we are able to accomplish maximum performance in today's dual and quad-core processors, then the performance level can reach phenomenal heights in the future when we start to see 100s and as many as a 1,000 cores on a single chip. As transistor sizes continue to drecrease to 65nm and 45nm and below, this optimistic assumption gets closer to reality.

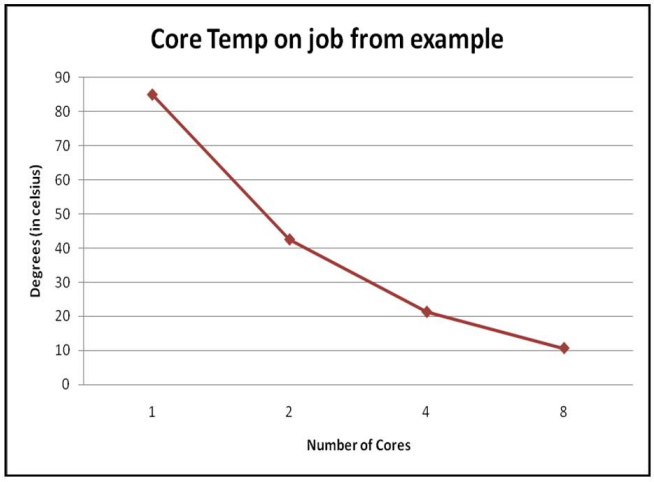


Fig. 1. Various temperatures of different core designs executing job from Example 1.

II. Thermal issues in Multi core designs

Thermal issues in multi-core designs have become important due to high transistor density in todays die size. This increasing amount of transistors and high performance demands are causing an increase in power which in turn, causes an increase in heat dissipation. This focus on heat has opened up a new area of study called *Dynamic Thermal management.* Dynamic Thermal management in multi-core designs is the regulation of Temperature in each core while a CPU is in operation. Dynamic Thermal management is useful when dealing with reliability, functionality, and cooling costs of a given design. A *hotspot* is, in general, an overheating core in a multi-core design. The main approaches used to tackle thermal issues in multi-cores have been *Stop & Go, Thread Migration(TM), and Dynamic Voltage and Frequency Scaling (DVFS).* All of these approaches have the objective to reduce overheating in cores; but each method varies in the amount of performance degradation. Thread Migration transfers the state of a overheating core to a core that is performing less work. A main source of high temperatures in todays CPUs is the frequency. Given that frequency is proportional to both power and performance, this makes it a difficult task to optimize. In DVFS, the frequency of a core is scaled down so that the core may cool down while still executing a program; the voltage is reduced to operate at the reduced frequency. In each technique, we will use something called a *reconfiguration interval*, which is a set interval that allows the maximum amount of cooldown [3]. Some of these techniques perform better based on the *thermal resistance* of the core. A good thermal solution (low thermal resistance or low-R) means that the cores have been fabricated with a good insulator, and uses low-conductivity material. Although a good thermal solution

may be good for thermal constraints, it may not be good for the cost of the overall chip. Therefore a cheaper and high-R solution may be used in most commercial multi core designs. We shall discuss these thermal methods in further detail in the following sections.

A. Stop & Go

The Stop & Go is the simplest of all the techniques, which makes it the easiest to implement. Stop & Go can be implemented on both global and local scale. Using the global approach, When any core reaches a critical temperature level, All cores are shutdown until non critical level has been reached. It is safe to assume that this technique is not used frequently by itself (unless the core is not is doing any work) being that it would stop throughput every time a core reaches a overheating temperature which is devastating to the performance level. If Stop & Go is implemented locally, then only the core that is overheating will be stopped until it has cooled down. Stop & Go techniques mentioned in [3] were clock gating the core, scale the voltage , and saving the state of the core and cutting the core off. Clock gating is an implementation where the clock is on only when it is in use. Using the clock gating method, it holds the state of the core, but still contributes to leakage power. The voltage scaling technique lowers the voltage to a much lower level (often called "sleep" level) where it cannot generate a logical output. The voltage scaling technique uses much less energy than an operating core, but the core is still consuming energy and not performing any work, which is wasted energy. The technique which completely stops the core, saves the current state of the core, and then shuts the core off completely. This technique generates no power which allows the core to cool down faster. If the state of the core is saved before shutting down, this method would be the better choice of the options.

Example 1: Suppose we are given a core that runs at a frequency of 1.5GHz and completes 1 thread in 5 clock cycles. The max temperature T_{max} for the core equals $100°C$ and the critical temperature T_{crit} equals $85°C$. It takes 5 threads to raise the temperature $1°C$. Find the number of threads it needs to run concurrently and the time it takes before the core hits its overheating point Assume that the core's initial temperature is $0°C$.

Solution: Given that it takes 5 threads to raise the temperature $1°C$, then we have

$$\frac{5threads}{1degree} \times 85degrees = 425threads$$

This is the amount of threads it needs to run to reach T_{crit} which is the overheating point for the core. Now we must calculate the time it takes to reach this point. Our first step is find out the number of *Clock cycles* it takes. Given

$$CPUclockcycles = instructioncount \times \frac{Cycles}{instruction} \quad (3)$$

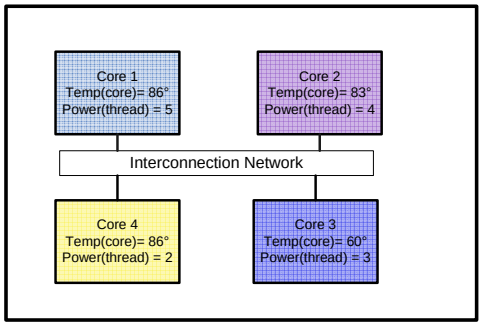


Fig. 2. Quad Core processor with core temperatures and thread's power consumption

we get

$$CPUclockcylces = 425 \times 5 = 2125clockcycles$$

From this we can compute the CPU time with the equation below

$$CPUtime = \frac{CPUclockcycles}{clockrate} \quad (4)$$

$$CPUtime = \frac{2125}{1.5GHz} = 1.42\mu sec$$

Although this example is unrealistic, we can get an understanding of how multi cores can overheat. In Figure 1 we show the temperatures of multiple cores running the same program as in the example. As illustrated, we can see a decrease in temperature as the number of cores increase. When cores are completely shut down, this has a negative(and the highest)effect on the overall performance. Thus using the method of Stop & Go alone would not be best for performance of a multi core processor today.

B. Thread Migration

***Thread Migration* is a thermal management tool used in multi-core processors to reduce a core's temperature by moving the high power-consuming thread to a different core and assigning a less power-consuming thread to the *hotcore*. Thermal issues are handled due to the work that can be distributed from an overheating core to the other cores so that an overheating core can be cooled down**

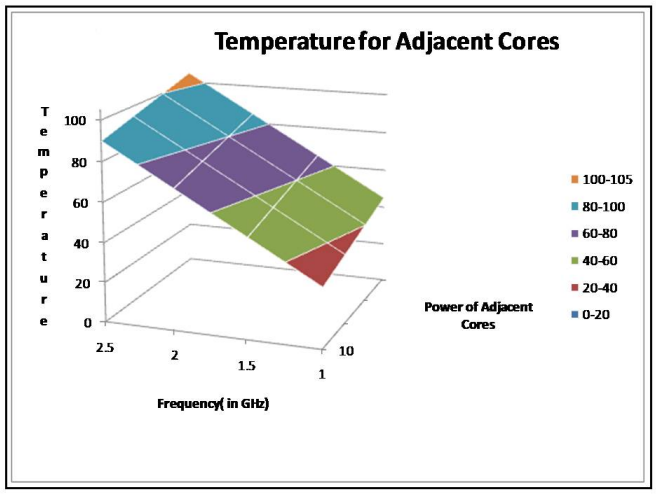


Fig. 3. Thermal surface of Adjacent Cores at different frequencies

by doing less work. *Temperature-based Thread Migration* swaps threads on hottest cores with threads on coldest cores. [4] uses a Counter-based Thread Migration technique where they arrange cores based on a temperature interval (This interval is based on the difference between highest critical limit and a second highest critical limit). In [3], he introduces two new approaches of *TM* which are *Improved Counter-based Thread Migration* (ICBTM), and *Power-based Thread Migration*. The *ICBTM* is a variation of the previous counter-based technique except the cores are arranged by their temperature and thread migration is not implemented if the temperature T_{core} is more than 1° under the critical temperature limit. The Power-based technique arranges the cores in increasing order by temperature and arranges the threads in decreasing order by their power consumption in the last reconfiguration interval. The most power-consuming thread will then be paired up with the core that has the lowest-temperature.

In Figure 2, we are given a Quad core processor with a core's temperature and the power consumption from the thread running on the core. Suppose that the critical temperature is $85\circ$. Using the *Temperature- Based Thread Migration* we switch threads running on cores 1 and 2 with the threads on 4 and 3 respectively. Using *ICBTM*, we get one thread migration between 1 and 4 because the remaining cores have degree less than one, thus TM is not executed. If we assume that the power values given for the threads were same in the last reconfiguration interval then we have a matching of Core 1 $\rightarrow$ Thread 4, Core 2 $\rightarrow$ Thread 3, Core 3 $\rightarrow$ Thread 2 and Core 4 $\rightarrow$ Thread 1. The advantage of *TM* over *Stop & Go* is that each core is still doing work at all times, therefore performance is only reduced by the context switch, which can be done within the reconfiguration interval, if the length is properly set.

C. Dynamic Voltage Frequency and Scaling

DVFS can be used to reduce thermal conditions on the global and local scale. With Global DVFS, voltage and frequency is reduced in all cores when one core reaches the critical temperature limit, so that the *hotcore* can cool down. In Local DVFS, the *hotcore* frequency and voltage are reduced when it reaches the critical temperature. Using Global DVFS, thermal issues can be resolved quicker than Local DVFS because Global DVFS reduces all of the core's frequency, thus reducing the overall temperature, but performance is effected by the lowering of all the core's frequencies. On the other hand, Local DVFS resolves thermal conditions, and has less effect on the overall performance. One research technique that has used DVFS is Physical aware frequency scaling. Physical aware frequency scaling implements DVFS and uses two approaches: Aggressive scaling, and Criticality-based scaling. Aggressive scaling reduces the frequency of the hotspot core, so that $Temp_{hotspot}$ is reduced in time interval *t*. The use of aggressive scaling lowers the temperature of the hotspot, but reduces the level of performance significantly as well. Criticality-based scaling reduces $Temp_{hotspot}$ as well as $Temp_{adjCores}$ so that performance does not have to be effected as much. If we use a fix range for the power of adjacent cores, $P_{adjcores}$ and the power of the hotcore, $P_{hotcore}$, we can generate temperature as a function of frequency(T(F)). By keeping a fixed range on frequency, F, and $P_{adjcores}$ we get the temperature as a function of $P_{hotcore}$ (T($P_{hotcore}$). If we use a distinct range of $P_{hotSpot}$ then we can illustrate temperature as $T(F, P_{adjCores})$.We now have a surface for temperature which can be illustrated by $T(F, P_{hotSpot}, P_{adjCores})$ where F is the *frequency*, $P_{hotSpot}$ is the initial power of the hotspot, and $P_{adjCores}$ is the average power of the adjacent cores. Fig. 3 gives an example of this surface.

Example: Suppose that we have the data in Figure 3 and we have a core at 80°. At what frequency should the core be scaled down to reach 40°, and what value should the power be for the adjacent cores?

Solution: . If we look at the given data in Figure 3, we can see that at 80°, the frequency is at about 2.5 GHz. Thus to get the temperature down to 40°, we need the frequency to be scaled down to about 1.5GHz. We can also note that the power of adjacent cores should be around 10W. This is a short example of how Physical Aware Frequency Scaling is used.

D. DVFS and Thread Migration

In [3] the idea of combining both DVFS and Thread Migration was introduced to lower performance degradation. Global DVFS and Local DVFS can be combined with the various techniques of *TM* to minimize the level of performance due to thermal management. Each combination has a different effect based on the type of

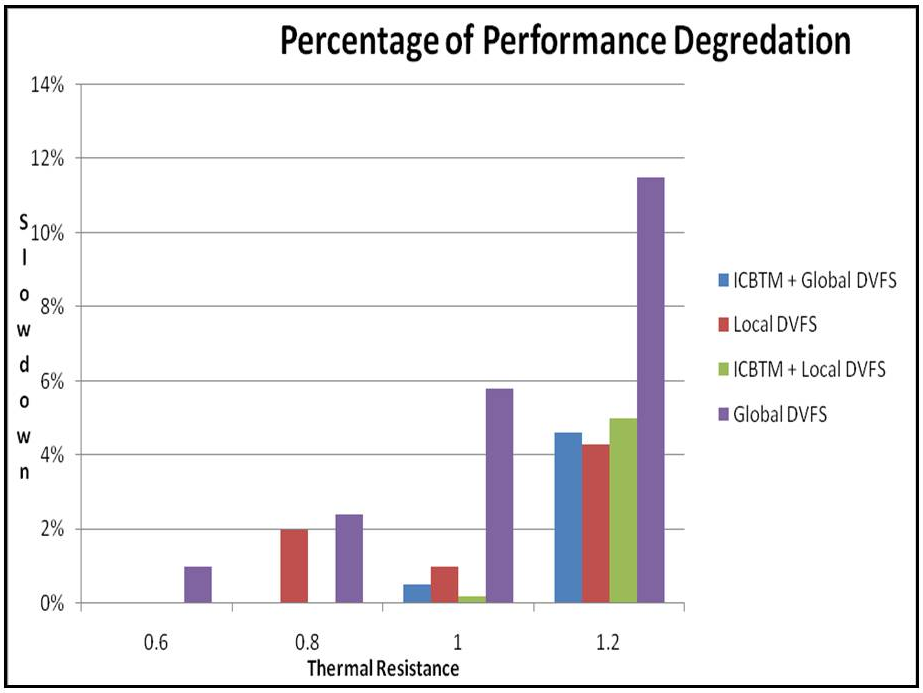


Fig. 4. Performance Degradation using Combined Schemes

thermal solution implementation. For low-R solutions, the ICBTM and Local DVFS performed the best. In normal thermal solutions, the combined schemes of TM and DVFS did not prove as useful. Local DVFS has least amount of performance degradation than any of the combined schemes. Global DVFS and TM generate low performance effects being that the number of migrations are small because Global DVFS eliminates the amount of thread migrations due to voltage and frequency scaling. Figure 4 gives a small portion of results found by [3].

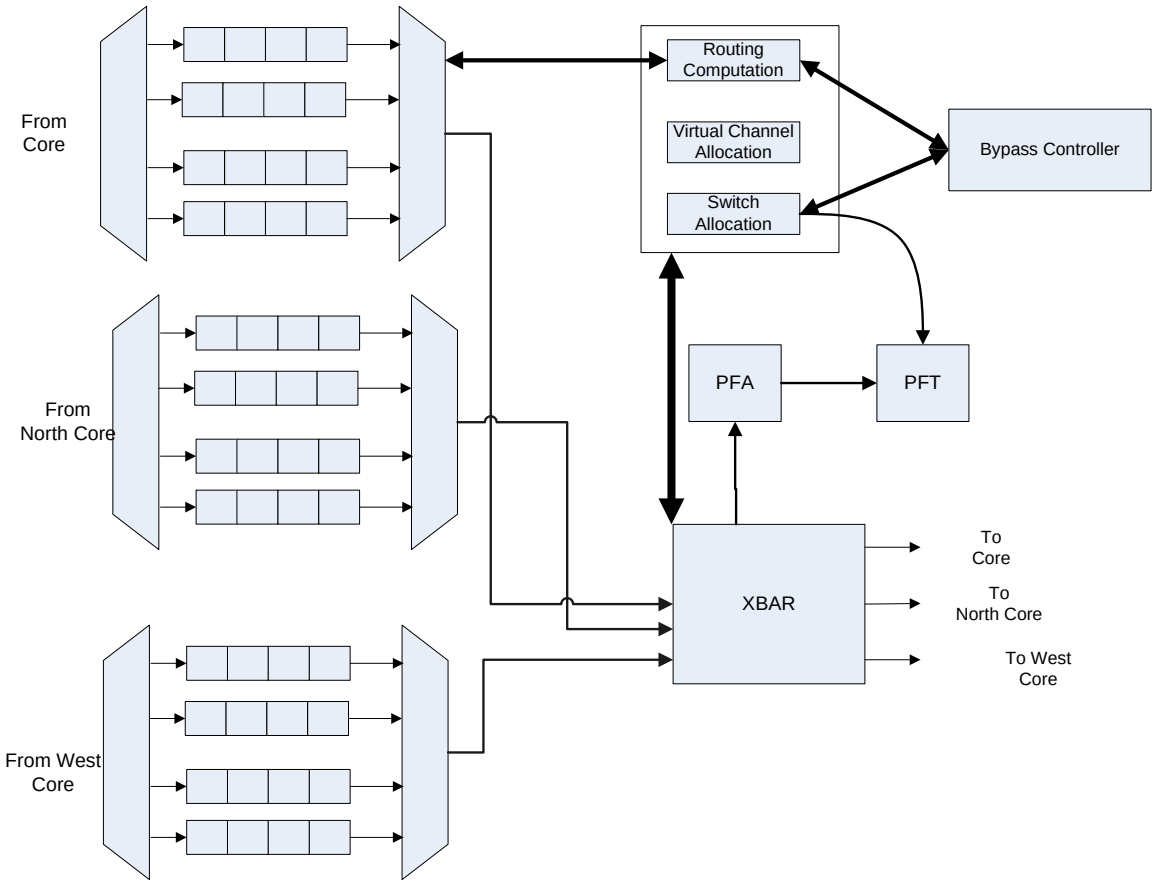
III. Interconnection networks and performance

Performance has always been viewed to increase by the number of cores and high clock speeds. But what if the communication between these cores was not very efficient One can ask "Are you getting the max performance out of that processor?". The answer would be, of course, no because interconnection networks and its relation to performance plays a major role in multi-core design. Interconnection networks can have an effect on performance(speed and latency), area, and power in a processor. Due to these effects, to create a maximal performance processor, the architect must design both cores and interconnection network together. Being that core communication can be viewed as a Local Area Network(LAN), researchers have created interconnections called *Network on a Chip* (NoC). This network's architecture is similar to a network that you may see in a lab. In recent years, many network topologies have been used for multi-cores, such as the *Torus Ring*, the *Crossbar*, and the *Hypercube*. The *Torus Ring* is a two dimensional topology that connects the cores horizontally and vertically, where the last core is connected to the first core in a ring fashion. An *Crossbar* topology is a frequently used topology where communication between any core can be accomplished in one transition. An *Hypercube* is a topology where the interconnection between cores is set up like a three-dimensional cube such that each adjacent core can be reached in one hop. Each topology differs in complexity, power consumption, and performance. These definitions define common terms used in interconnection networks:

1) ***Arbiter*** **- an electronic device that allocates shared resources.**
2) ***Repeater*****- is a device that strengthens a signal that has to go a great distance through regeneration, much like a buffer.**
3) ***Packets*** **- Information grouped together to be sent at one time.**
4) ***flit*** **- flow control digits located on packets. Smallest unit that can perform flow control.**
5) ***IP core*** **- intellectual property core used to create FPGAs and ASICs.**

A. Fast Path

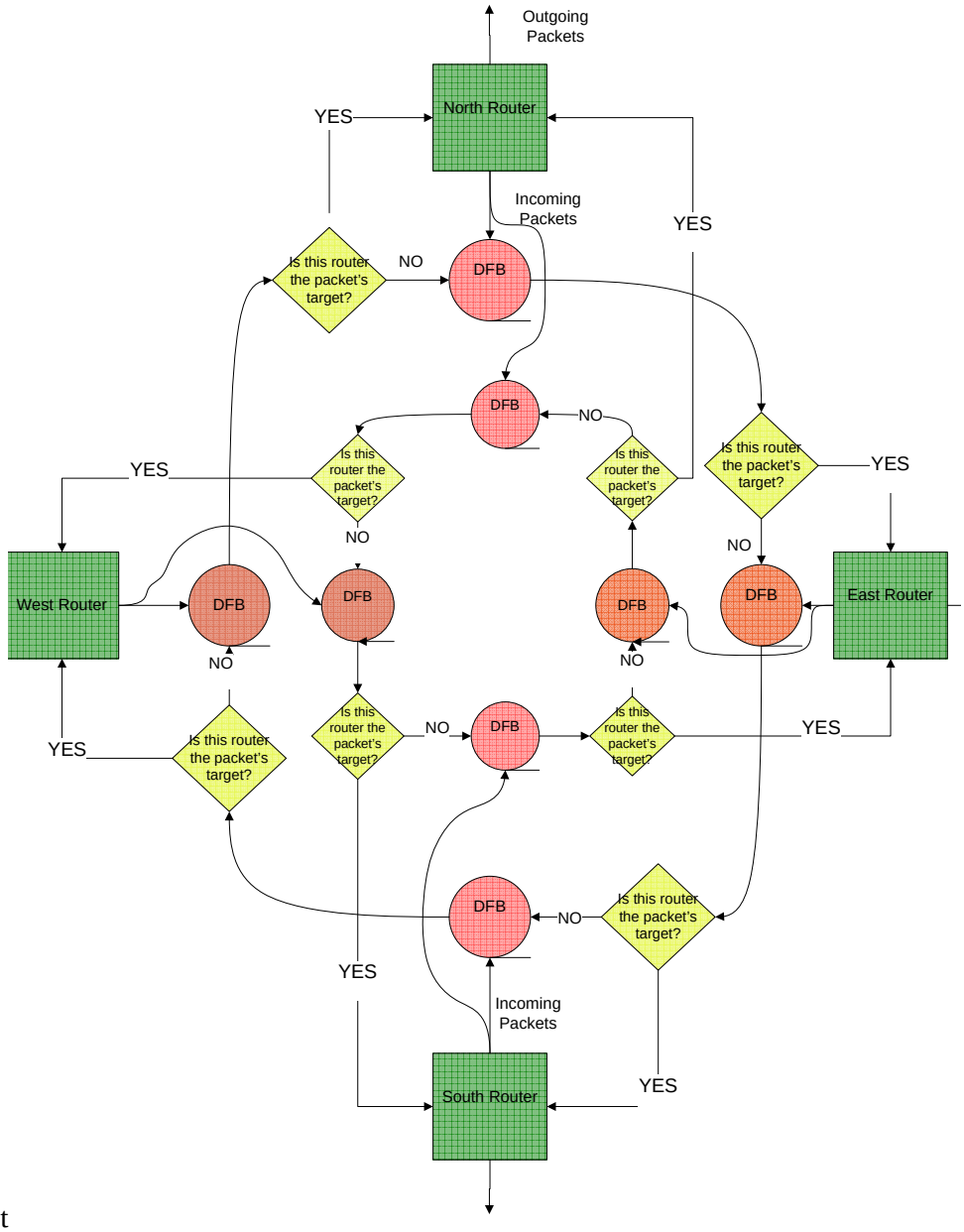
The *Fast Path* is an interconnection architecture proposed by [5] used to send flits through a network of cores efficiently using a priority scheduling for *fast paths* in a



heightheight

Fig. 5. Design of the Fast Path Architecture

router pipeline on a NoC. The Fast Path architecture uses temporal locality to give higher priority to those paths frequently used. According to [5] a generic NoC router pipeline can have two stages, one for the parallel execution of *router computation, virtual channel allocation, and switch allocation*, the other stage for *switch traversal*(Crossbar). Figure 5 gives an design model of the Fast Path Architecture. Virtual channels are used in a similar fashion as virtual memory, where each virtual channel is mapped to a physical channel in the network. The switch allocation uses an arbiter to decide which channel will be granted access to send information. The output of the switch allocation is then sent to the Crossbar. The additional area and power overhead in this architecture is due to a *Path Frequency Analyzer*, a Path Frequency Table and a *Bypass Controller*. The *path frequency analyzer* counts the amount of transfers in each path. The *path freqency table* keeps record of whether a path is Fast Path or not. If the path is a 'Fast Path' then table stores a logic 1 for that path; it stores 0 otherwise. If a path has more than $t_{threshold}$ flit transfers, the path is listed as *Fast Path* and the *path frequency table* is updated to 1 for that path. This path is given priority over normal paths, but a limit is implemented to prevent starvation in the normal paths. According to [5] using the *bypass controller*, a fast path can obtain a single stage router pipeline by executing the second stage of a current instance, and the first stage of a second instance in parallel. This one stage pipeline can be done only if two fast path instances occur back to back. The Fast Path Architecture has shown to increase performance by up to 30% in synthetic benchmarks due the use of *Fast Paths*; performance increased by up to 20% in real benchmarks. [5]. [5] shows that the Fast Path's best performance in random traffic were smaller packet sizes, while larger packet sizes performed better on transpose matrix traffic.



heightheight

Fig. 6. Design of the Rotary Router. Note: For spatial purposes, the Demuxs are omitted from the Routers

B. Rotary Router

The Rotary Router architecture is a on-chip network that allows for information to be sent in a clockwise and counter-clockwise fashion on independent ring paths. A Flow design is given in 6Each ring path is made up of Dual-Port FIFO Buffers (DFB) which allows the packets to enter/exit a ring path and/or move along a ring path as well. Based off the architecture, [6] states that the ring structure avoids deadlocks, Head of Line Blocking (When a packet in the front of a FIFO structure blocks the remaining packets behind it), and is flexible to many topologies. This router consists of three stages: the *input,buffered segment and output*. In the input stage, a demultiplexer is used to determine which ring the packet will use, but the idea is to choose the direction that is closest to the packet's destination. The buffered segment of the router determines whether the packet stays on the ring, or whether it has reached its destination and ready to go to the output stage. The output stage sends packets to the neighboring router using a multiplexer to choose which ring path the packet is coming from. From [6], this architecture had over 50% increase in performance when compared to a Bubble router due to its increase in performance on small packet sizes.

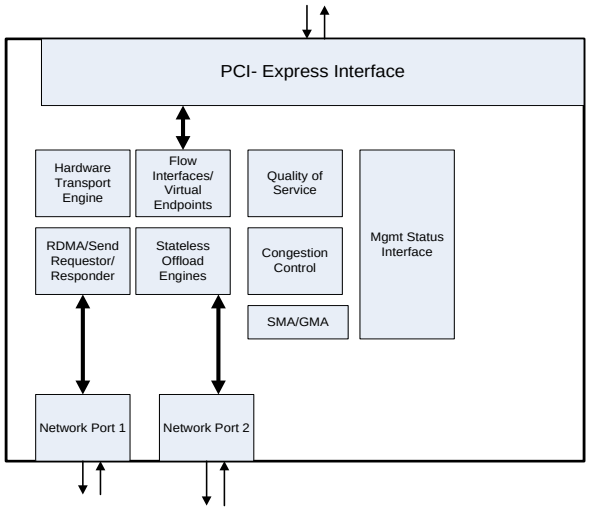


Fig. 7. Structure of the Host Channel Adapter in ConnectX Architecture

C. Interconnection Architectures used in today's commercial processors

The Mellanox ConnectX Infiniband Architecture is the latest of Mellanox's Host Controller Adapter for servers. Figure 7 gives a block diagram of the structure of ConnectX HCA. This arhcitecture processes incoming packets and sends them to their respected destinations where it can be another HCA, or a connected port. The ConnectX architecture has fast scheduling engines that allow for it to assign duties to idle cores in a multi-core processor. This allows for scheduling to be done completely in hardware. ConnectX has a reduce latency difference in RDMA-Write(Remote Direct Memory Access) and Send/Transport modes by $0.2\mu sec$. The ConnectX uses a Scalable Reliable Connected (SRC) transport which allows for one connection to be necessary for all processes in a core processor. [7] stated that the ConnectX increased bandwidth over the previous generation Infinihost III by a factor of 10 in small and medium packages.

The IBM C64 crossbar is used in the *IBM Cyclops 64 Architecture* The Cyclops 64 Architecture has up 80 processors on one chip with two cores within each processor, giving it a total of 160 cores. Given this large amount of cores, a fast interconnection is needed to control data flow. The IBM C64 crossbar architecture uses a 96 x 96 buffered crossbar topology to route information through multi-cores, caches, I/O devices and SDRAM memory banks. Each port connected to a processor has a *source control unit,* a *target control unit,* a 96-1 *multiplexer*, and a *data queue* (FifoD7). Information is sent using a Source control unit and Target control unit between two ports. The source control generates control information, manages data buffers in the FifoD7 and send requests to the Target Control of the destination port [8]. The target control unit selects the winner of requested source ports using the arbiter and sends a token back to the winning port. A 96-1 multiplexer is used to create the crossbar between all ports. The data queue is combined with the Mux to create the data path for the crossbar design. Results have shown that a channel can created between any two ports with a minimum latency of 7 cycles. Through the use of virtual channels, data can be sent in both directions in parallel. Based on results from [8] the saturation point for latency was reached at 0.6 packets per cycle with uniform random traffic and throughput remained constant once it reached it saturation point(proves that is does not contain blocking).

The Element Interconnect Bus(EIB) is the interconnection network used on the *Cell Broadband Engine Architecture*. The EIB transfers data through the following connections: eight Synergistic Processing Elements(SPE), one Power Processing Element(PPE), one memory interface controller, and one bus controller. Each connection has an Bus Interface Unit to interact with the EIB. The *EIB*

consists of four 16-byte wide data rings, a shared command bus and a central arbiter [9]. Data rings are used to transfer info throughout the main components. Each data ring can hold up to three transfers simultaneously with no overlapping. The command bus gives out commands and takes care of coherency throughout the network. The ideal throughput for the EIB is 307.2 GB/sec, but due to limitations described in [9], such as command bus design and data ring topology, the maximum throughput for non coherent data is 197 GB/sec, and 78 GB/sec in coherent data.

IV. Memory issues and Performance in Multi-cores

The speed of memory and processors have a well known relation which is commonly called the *Von Neumann gap*. This *Von Neumann gap* states, in general, that the processor speeds up 60% every two years, while the memory only speedups by 10%. This gap is crucial to the performance of the overall CPU; even the fastest CPU is slowed down by slow memory accesses. In multi-core designs, there are issues with *memory hierarchies*, *coherency*, and latencies. All these factors have a effect on the performance of a design. A *memory hierarchy* is a structure in which faster memory is smaller and located closer to the CPU, while the slower memory are larger and further away from the CPU. Most of today's multi-core's memory architectures have two levels of cache memory on chip, and the remaining memory off chip. The first level ,L1, cache is usually separate from each core, and the second level, L2, is usually shared by all the cores. Memory hierarchies commonly use the *inclusion property* where all data in an upper level memory is copied to the lower memory level beneath it. This inclusion property keeps coherency within lower levels of memory, but in multi-cores where each core has its own private L1 cache, we must assure that data stored in these private caches are up to date. Techniques such as *snooping protocol* are used to tackle this measure. The *snooping protocol* allows for the private L1 caches to constantly check a broadcasting bus(this bus connects all L1 caches to each other and to the L2 cache as well) to see if any changes have been made to the data located in memory. *Latency* is usually effected when accessing memory off the chip in a multi core processor. Accessing memory of the chip can account for more than 2x the amount of latency created by L2 memory access [15], therefore we want to minimize the number off chip memory accesses. Various cache techniques are researched to solve this problem.

A lot of focus has been given to the performance of the L2 cache in core designs. In most architectures,the L1 cache is exclusively owned by a particular core processor while the L2 cache is shared among all cores. [13] investigated the robustness of the partitioning of L2 cache to tasks, versus using the standard shared method. The idea is to allocate a certain amount of L2 cache for a given task, T, running on a processor P. Results from [13] show that the proposed partitioning of L2 cache is 10 times less sensitive than the popular shared L2 cache design. [14] researched the use of μcaches, as a substitution for the normal L1 cache, to reduce performance degradation created by cache latencies and area issues in a core design. μcaches are defined as small caches attached to each core in the 1st level of an memory hierarchy, and each μcache is attached to a shared instruction cache. Results by [14] show a large difference in area versus modern cache sizes, but very little improvement in performance. Results show that this technique maintained very high constraints in its simulation, and used single threaded processors to develop an increase in performance. This may lead to the conclusion that this method is not very promising for the future of multi-core, given the direction that we are heading in today.

A. Cache management using a L3 cache

A cache management method has been designed to manage write backs from L2 caches to a off-chip L3 cache to better performance. The L3 cache is an off chip victim cache where modified lines can be stored but has a less access latency time than main memory. [15] has proposed two methods of cache management to help benefit performance of the chip. The first technique attempts to limit the number of cache lines written back to L3 that have not been modified. The second approach writes cache lines to a neighboring L2 cache, allowing for the transfer to occur on-chip rather than taking the miss penalty of writing back to the off-chip L3 cache. The motivation of the idea to limit *clean write backs* comes from the performance degradation due to the increase in network traffic. Clean write backs can be defined as data written back to a lower level of memory that has not been modified. If a great number of loads have to access the off-chip memory and there is also a high number of clean write backs, these *clean write backs* are creating unnecessary traffic on the network. This selection scheme first uses a filter in the L2 cache that determines whether a evicted *clean* (not modified) line should be sent to the L3 cache or to main memory. This approach uses a Write Back History Table, where entries are allocated by clean write backs from the L2 cache given that the cache line is located in the L3 cache. After the cache line is brought back into the L2 cache and the L2 cache is ready to replace it again, if the cache line remains clean, then the WBHT is checked to see if the line is still a valid entry. If this is true, then the write back to the L3 cache is not carried out, because a valid copy of the cache line already exists in the L3 cache. There are drawbacks to the WBHT such as the potential of inconsistent data with the L3 cache and a performance penalty for mispredictions when the WBHT is not needed. As a solution for the second case, [15] uses a timer and

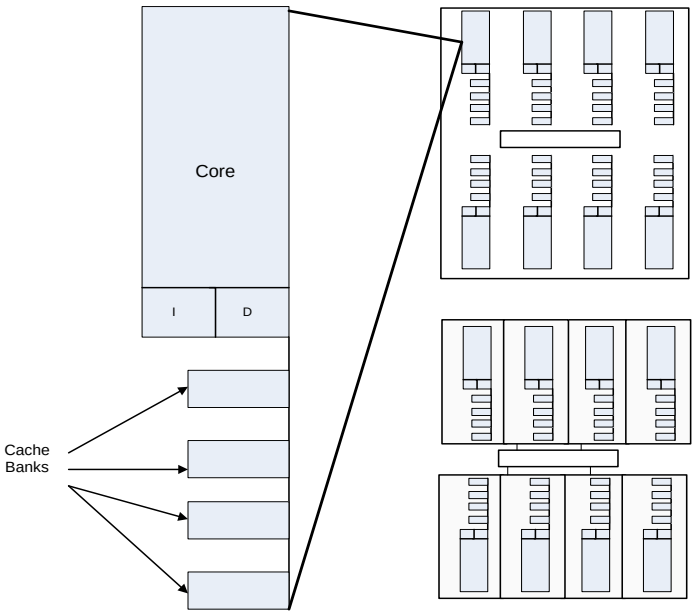


Fig. 8. (left) Design of the core. (top right) 8 core processor with a sharing degree of 8. (below right) Processor with a sharing degree of 1.

a timer which detects when it is more efficient to write all the clean cache lines back to the L3 cache. Given a number of retries, $count_{retries}$ due to race conditions and a given time interval, *t*, if $count_{retries}$ goes below a number, $count_{threshold}$, the WBHT is switched off.

The second technique writes back to a neighboring L2 cache. For this to occur, this cache line should have a high probability to be reused. This can be determined by a small table implemented to keep track which cache lines have been written back and reused by L2 caches.When a write back to a neighbor L2 cache occurs, the ideal cache line to replace on the "retrieving" L2 cache would be an invalid cache line; the next best choice would be a "shared" cache line. "Exclusive" and "modified" cache lines are avoided for the sake of a future miss penalty. Performance using the first approach improved a maximum of 10%. The best performance results occurred in high network traffic between the L2 and L3 cache. In [15], the simulation results showed that the WBHT table was correct 60% - 75% of the time. Performance increase using the second method was overall minimum. Combining the methods had very little effect on performance increase as well.

Example 1: Lets say we are given Table I. Suppose the cache lines were placed on the WBHT in increasing order starting from index 0 going up to 4.

1) **What actions occur when the cache line with the Tag data of 100_{bin} is placed on L2 cache and about to be replaced again?**
2) **What actions occur when the cache line with the Tag data of 001_{bin} is placed on L2 cache and about to be replaced again?**
3) **Lets say a new cache line needs to be placed on the WBHT, in what position will it reside?**

TABLE I
WRITE BACK HISTORY TABLE

Index	Valid	Tag	Data
0	1	010_{bin}	0100_{bin}
1	1	011_{bin}	1011_{bin}
2	0	001_{bin}	0101_{bin}
3	1	100_{bin}	1111_{bin}
4	0	101_{bin}	0001_{bin}

Solution: When the cache line with Tag data 100_{bin} is about to be replaced, the L2 cache checks the WBHT to see if this cache line exists in the table. We see that this cache line is located in Index 3, therefore the write back to the L3 cache is terminated. For the cache line with Tag data 001_{bin}, when looked up on WBHT, we notice that the cache line was located in Index 2, but the *Valid bit* is set to 0, therefore the cache line is not present. Thus this cache line is written back to the L3 cache. If we wanted to place a new cache line on the WBHT, then we would have to eject one of the present entries. Assuming that *Least*

Recently Used **replacement policy is used, then the entry located in Index 0 would be ejected. Thus the position of the new cache line would be Index 0.**

TABLE II
AVERAGE NUMBER OF RETRIES AND CLOCK INTERVAL FOR APPLICATIONS

Application	Retries	Clock Cycles
ransford	3500	1×10^6
morel	4000	2.5×10^6
hyman	2500	1×10^5
junior	5000	3.5×10^6
cameron	6000	4×10^6

Example 2: Suppose that we are given information from Table II. Given this data, what would be a good time interval and a good value for $count_{threshold}$?

Solution: We want to find a good median for both values so that we don't switch the WBHT off too early or too late. By averaging the values for the *retries*, we get

$$\frac{3500 + 4000 + 2500 + 5000 + 6000}{5} = 4200$$

Thus a good value for $count_{threshold}$ would be 4200. Now we need to find the average for the clock cycles. This gives us

$$\frac{(1 + 2.5 + 0.1 + 3.5 + 4) \times 10^6}{5} = 2.2 \times 10^6 cycles$$

So a good interval for switching the WBHT off would be 2.2×10^6 clock cycles after 4200 retries.

B. Non-Uniform Cache Architecture(NUCA) for Cache Sharing

Research done by [16] involves finding the best cache sharing organization that improves the performance in a multi core processor. Much emphasis is placed on the L2 cache. The L2 caches in most multi-core are commonly shared. The shared caches can be arranged in two different designs: one design where all of the cores share their L2 caches, and one design where the shared caches are logically partitioned for each core(or many cores) to have a equal portion. A *sharing degree* can be defined as the number of processors sharing their L2 cache. The *Non-Uniform Cache Architecture* permits flexibility in the selection of a sharing degree. Lower sharing degrees are better for hit latencies, while higher sharing degrees are better for hit rates. Higher sharing degrees reduce miss rates by having a larger area of partition so that more cache lines may be stored and by minimizing the number of duplicate copies of cache lines. In Figure 8, we can see how *cache banks* are used in the L2 cache. The L2 cache is arranged into equal cache banks where each processor has direct access to each bank inside its L2 cache, and access to others by a switch network. Each processor has its own seperate set of cache banks. In Figure 8 on the right an illustration of the sharing degrees of 1 and 8 are shown for an 8-core processor. The degree of sharing can be changed by adjusting the bits used to route memory addresses to a particular cache bank. If the bit string for routing a memory address is the same for all *n* cores and the bit string routes the address to the same cache bank, then the sharing degree equals *n*. A issue that must be taken care of when dealing with shared caches is the coherency between L1 and L2 caches. For L1 coherency, the *directory based coherence protocol* keeps coherency by placing the status of the data inside the L2 tag information. The L2 tag holds information about which L1 caches holds copies of the designated cache line. When a L1 cache sends a signal to the L2 cache for an update, the L2 cache sends invalid signals to all L1 caches containing the cache block. For L2 cache coherency, a centralized directory keeps information about all of the L2 caches. This central directory determines whether to get a cache line from a nother L2 cache or to go to off chip memory on a L2 cache miss. This coherence protocol was used instead of the snooping protocol, because it detects on-chip cache misses faster. A *Static mapping* (S-NUCA) places blocks in a designated cache bank. The lower bits of an address gives the position of the cache bank where the data should be stored. A *Dynamic mapping* (D-NUCA) is used so that a cache block has the option to reside in more than one bank(not at the same). D-NUCA allows for frequently used cache blocks to be placed closer to the CPU, thus giving it faster access time.

The first migration policy inputs cache blocks at the bottom of the bank array, and only moves blocks vertically along the bank array(D-NUCA in 1 dimension). The second migration policy places cache blocks in any L2 cache bank set in the array(2D). New blocks are inserted at the cache bank set nearest to the processor. In multi-cores, blocks could be signaled to migrate from two different cores, causing it to move back and forth. To prevent conflicting migrations, a counter is kept in the cache tag information so that a cache block will only migrate in a given direction based on a count that acknowledges the given direction is *saturated*. To look up a cache block [16] uses a distributed partial tag to search the blocks located in each column using a partial tag array. In the one-dimensional D-NUCA, partial tags are used to find which column the block is located. If a cache block is not found in the first bank, then it searches other banks within that one cache bank set. In the two-dimensional D-NUCA, it uses the partial tag to search all possible columns the cache block could be located. The column chosen first is the one nearest to the processor. Results have shown that the best configuration for the L2 cache for performance has a sharing degree of 4.

V. Parallelism and its effects on performance

Parallelism has been one of the biggest breakthroughs of performance. Performance can be viewed, in a nutshell, as how fast it takes to complete job. Parallelism allows us to increase this performance, because it allows instructions to execute simultaneously. *Instruction-level Parallelism* was introduced first where the number of instructions executing at the same time was the target. Techniques used to accomplish ILP have been *pipelining* and *Tomasulo's Algorithm*. *Pipelining* issues instructions into a multicycle design and executes them cycle by cycle in a number of stages. Given a pipeline with *n* number of stages, there can be at most *n* of instructions in the pipeline at the same time. A fallacy with pipelines is when a missed branch prediction occurs. When missed branch occurs, then the instructions that were executed after it have to be flushed out of the pipeline which can have a long latency, depending on how many instructions came after the branch. *Tomasulo's Algorithm* is an hardware algorithm that was implemented to execute floating point instructions in the late 1960s. This algorithm allowed out-of-order execution and prevented write after read (WAR) and write after write(WAW) through register renaming. The algorithm contains three stages in general: an issue stage an execute stage and an write back stage. In the issue stage, WAW and WAR hazards are handled using register renaming. In the execute stage, instructions are sent to the logical units to carry out their specific operations. One execution has completed, the value from the execution is broadcast on a common data bus. In the event that a operation was waiting on the particular value, this operation is stored in a reservation station. Once the value that the operation has been waiting for has been broadcast, then that operation is carried out. This technique is still used today to increase parallelism in CPUs.

The another breakthrough in parallelism comes from the technique of *Thread-Level Parallelism*. *Thread-Level Parallelism* is a software technique where a program can be broken up into threads(small process that uses some shared resources) and executed simultaneously in a synchronized manner. Sychronization techniques used for scheduling threads are *Round Robin* and *Priority Scheduling*. Round Robin is a scheduling technique where each thread gets the same amount of time to run on the CPU. Priority Scheduling gives threads a priority on the order in which to run the threads. The priority value is decreased overtime to prevent starvation amongst other threads. *Starvation* is the event that one thread is given all of the CPU time to where no othere threads are getting executed. Another issue with threads is called *race conditions*. Race conditions occur when two threads needs access to the same shared resource. If race conditions are not avoided, it could lead to *deadlock*. The technique of TLP actually increases the amount of performance, but is it the best that we can do? The answer would be not at all. The complexity in avoiding deadlocks in large amounts of threads has hinder our maximum performance level in today's multi-core processors. This issue has been avoided for the most part, and for increase in performance, we have turned to adding more hardware. What happens when we reach that point in hardware, when we can't add any more transistors onto a single die? Although there has not been a breakthrough in this problem, there have been other techniques researched in the research community. The main technique research is generating parallel code through the compiler. We explain this technique in the method used below.

A. Software Pipelining

The software pipelining method used in [17] transforms nested loops into parallel executable code that can run on a multi-core processor. This technique is used at the compiler level, therefore programmers do not have to change their code. The Single Software Pipeline(SSP) chose the most beneficial outer loop for pipelining and ran the inner loop iterations sequentially. Lets say that we have a interval T, which is the interval in which we can issue another outermost iteration. The interval is dependent on resource availability and dependencies. The SSP takes a loop nest of *n* as its input, chooses the best outermost loop, and then issues the outermost loop every *T* cycles. We note that a *kernel* is an instance where each operation in the code is located exactly once. A *subkernel*, S_i is similar to the kernel except it applies only to a inner loop. Operations are scheduled in a manner where there are no resource conflicts or data dependencies. To avoid resource conflicts, the software pipleine creates a delay after every S_n different outermost iterations, if necessary. Example: Suppose we are given the the following code:

```
For i = 0 to 2
{
op1
For j = 0 to 1
{
op2
op3
}
}
```

Suppose there is a data dependency between *op1* and *op2*. There are two functional paths available. Illustrate how the Software Pipeline would execute this code.

Solution: For this code, we get the following results from Table III.

Looking at Table III, we can see that there are no data dependencies so there is no need for any additional stalls besides the ones that were given. Also we can note that code was able to executed within the two functional paths.

This "Multi Threaded Software Pipeline(MTS)" takes loop iterations of nested loops and runs different sections

TABLE III
ITERATIONS OF FOR LOOP

Cycles	Iteration 1	Iteration 2	Iteration 3
1	op1		
2	nop		
3	op2	op1	
4	op3	nop	
5		op2	op1
6		op3	nop
7			op2
8			op3

on different cores. Memory dependecies are also possible therefore memory should be accessed in the correct order. For the MTS to be complete it must maintain order and synchronization to maintain data dependencies. MTS schedules each group of S_n outer loops to a different thread unit in a multi core. Signals are issued before and after repeating patterns to make sure code is kept in order. Every thread unit contains a *wait* operation except the first. Synchronization is maintained by two counters: a synchronization counter keeps track of the number of synchronization signals and a clock counter which counts the progress of the thread unit(located on the core). A thread is only allowed to continue execution after a *wait* instruction if the synchronization counter is greater or equal to the clock counter. Wait instructions are stored in a scratch-pad memory(small high speed memory located on the chip) so the thread unit may have fast access to it. Signals are sent to the next thread unit after the execution of the last outer loop, regardless if that has completed or not. When all thread units have finished executing, the last thread unit sends an additonal signal to the first thread unit, which returns back to the main program. *Cross iteration regesiter dependencies* are dependencies located on different threads, which means they occur on different iterations. To resolve cross-iteration register dependencies, the dependence is transformed in to a memory dependence. A copy of the value is placed on a buffer in the scratch pad memory of the destination thread unit. The value is then restored using is restored using a local memory pad. The MTS was tested on the Open64 Compiler by IBM. Results have shown MTS to be scalable for up to 100 cores. The execution time speedup in [17] shows that the speedup increases linearly as the number of the thread units increase and as the program size increases. Exection time in MTS is effected by the factor created by cross iteration dependencies and initialization costs.

VI. THE FUTURE OF MULTI-CORE

In today's multi-core processors, we see designs with Quad-core and even 8-core processors(Cell Processor). Most research explained earlier used at least a 16-core processor as a test for their simulations. This lets us know that the increase in number of cores is promising in the future. Intel has duplicated a chip with 80-cores on board already, just to see if it was possible. [18]. [8] describes how IBM is building a petaflop computer with a million cores. There is also the introduction of the 45nm microprocessor using high-K metal gates where *K* is a dielectric constant that are used in characterizing capacitors. These high-K metal gates have been stated to reduce leakage power in the processor. The leakeage power has just recently become a big factor due to the small transistor scaling.

Software, unfortunately, has not been making such fast advances. There have been researchers who have generated parallel code through the compiler (i.e. Software Pipelining) but the hunt for creating parallel algorithms and programming remains the same. Even for multi-cores today, programming multiple cores has been proven to be difficult task. Once this breakthrough occurs, it will be a big benefit for both software and hardware. Software Engineers will have an easier job programming on multi-core processors, and Hardware designers will have better utilization of their designs.

REFERENCES

[1] http://en.wikipedia.org/wiki/Intel_4004

[2] Mukerjee, R. Memik, S. "Physical Aware Frequency Selection for Dynamic Thermal Management in Multi-Core Systems". *ICCAD'06* 2006, pp. 547 - 552.

[3] Chaparro, P.,Gonzalez, J., Magklis, G., Cai, Q., Gonzalez, A. "Understanding the Thermal Implications of Multicore Archtectures". *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS* Vol.18, Iss. 8, Aug. 2007, pp. 1055 - 1065.

[4] Donald, J., Martonosi, M., "Techniques for Thermal Multicore Management: Classfication and New Exploration". *Proceedings of International Symposium on Computer Architecture* 2006 pp. 78-88.

[5] Park, D. Das,R., Nicopoulos,C., Kim,J. "Design of a Dynamic Priority-based Fast Path Architecture for On-chip Interconnects" *IEEE Symposium on High Performance Interconnects*, Aug. 2007, pp. 15-20.

[6] Abad,P., Puente, V., Prieto,P., Gregorio,J., "Rotary Router: An Effiecient Architecture for CMP Interconnection Network". *Internation Symposium for Computer Architecture*, Jun. 2007 pp. 116-125.

[7] Sur, S., Koop, M., Lei, P., Dhabaleswar, K.,"Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand

Architecture with Multi-Core Platforms", *HOTI '2007*, Aug. 2007, pp. 125 - 134.

[8] Zhang, Y.P., Jeong,T., Chen, F., Wu, H., Nitzsche, R., Gao, G.R. "A study of the on-chip interconnectionnetwork for the IBM Cyclops64 multi core architecture. *IPDPS '06*, April 2006.

[9] Ainsworth, T.W, Pinkston, T.M., "On Characterizing Performance of the Cell Broadband Engine Element Interconnect Bus", *NOCS 2007*, May 2007, pp. 18-29.

[10] Held, J., Bautista, J., and Koehl,S., " From a Few Cores to Many: A Tera-scale Computing Research Overview", *White Paper: Research at Intel.Intel Leap ahead*, 2006.

[11] Ames, Ben. "Intel Tests Chip Design With 80-core Processor". Feb. 11, 2007. Oct. 9, 2007http://www.pcworld.com/printable/article/id,128924/printable.html.

[12] Parkhurst, J., Darringer, J., Grundmann, B., "From Single Core to Multi-Core: Preparing for a new exponential", *Proc.Computer-Aided Design*, Nov. 2006, pp. 67 -72.

[13] Molnos, A.M., Cotofana, S.D., Heihiligers, M.J.M., van Eijndhoven, J.T., "Static cache partitioning robustness analysis for embedded on-chip multi-processors", *Conference On Computing Frontiers*, May 2006 pp. 353 - 360.

[14] Becchi, M., Franklin, M., Crowley, P. "Performance/Area Efficiency in Chip Multiprocessors with Micro-caches", *Proceedings of the 4th International conference on Computing frontiers*, May 2007, pp. 247-258.

[15] Speight,E, Shafi, H., Zhang, L., Rajamony, R., "Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors", *ISCA '05* June 2005, pp. 346 - 356.

[16] Huh, J., Kim, C., Shafi, H., Zhang, L., Burger, D., Keckler, S."A NUCA Substrate for Flexible CMP Cache Sharing", *ICS'05*, June 2005, pp. 31 - 40.

[17] Douillet, A., Gao, G., "Software Pipelining on Multi Core Architectures", *PACT '07*, Sept. 2007, pp. 39 - 48.

[18] http://www.news.com/Intel-shows-off-80-core-processor/2100-1006_3-6158181.html.