10.1515/9781450361002-3 3 Architecture: Heterogeneity in Design 33 62

[Ogawa and et al. 1988, Archibald and Baer 1986, Eggers and Katz 1989, Moshovos 2005, Tomasevic and Milutinovic 1993]. That path is now targeting multicore architectures [Cheng et al. 2006, Ekman et al. 2002, Hammond et al. 1997, Krishnan and Torrellas 1999, Nayfeh 1998].

3.1 Memory System

The second path taken in memory system research is the cache hierarchy for multiprocessor systems. The research in that area consists mainly of coherence protocol schemes [Agarwal and et al. 1988, Archibald and Baer 1986, Eggers and Katz 1989, Moshovos 2005, Tomasevic and Milutinovic 1993]. That path is now targeting multicore architectures [Cheng et al. 2006, Ekman et al. 2002, Hammond et al. 1997, Krishnan and Torrellas 1999, Nayfeh 1998].

These two paths in memory system research proceed in parallel. But with the introduction of multicore architectures (AKA chip multiprocessors or CMPs), the two paths started to converge [Beckmann and Wood 2004], and the main theme was the high-bandwidth available on-chip, as opposed to the limited, and more expensive, off-chip bandwidth. However, wire delay represents a big challenge because it does not scale with the transistor [Agarwal and et al. 2004, Beckmann and Wood 2004]. This leads to the introduction of network on chip [Benini and DeMicheli 2002, Dally and Towles 2001, T. T. Ye 2003], which uses the experience gained at designing packet switched networks to design on-chip networks that connects cores and caches. With the increase in the number of on-chip cores and the fact that most of the current designs still make use of the traditional design of private L1 cache for each core and a shared L2 cache among cores, the problem of sharing the L2 caches among cores becomes important both in academia[Kim et al. 2004b, Qureshi and Patt 2006], and in industry. The problem of cache sharing for multicore architectures uses experience gained from cache partitioning in clustered architectures [Farkas et al. 1997]. Another trend is to use small caches to act as an aggregate large cache [Chang and Sohi 2006, Varadarajan et al. 2006]. These methods have the advantage of saving power but do not solve the problem of application interference in caches, where two applications have conflict cache requirements which happens very frequently in simultaneous multithreading architectures [Lo et al. 1997, Tullsen et al. 1995], multicore architectures [Borkar et al. 2006, Broadcom Corporation 2006, Ramanathan 2006, Sinharoy et al. 2005], and system built as hybrid of the two like Niagara (ULTRA SPARC T1) [Kongetira et al. 2005].

Another line of research that has been targeting single core and has migrated, *without much enhancements or change at first*, to multicore is cache replacement policy [Al-Zoubi et al. 2004, Guo and Solihin 2006, Jeong and Dubois 2003a, Megiddo and Modha 2004, Qureshi et al. 2006, Reineke et al. 2006, Wong and Baer 2000]. Cache replacement policy decides which block to evict to make room for a new incoming block.

Cache replacement starts to be evolve with the widespread usage of multicore and manycore processors [Guo et al. 2013, Pekhimenko et al. 2015, Ros et al. 2015]. However, cache hierarchy nowadays needs to deal with different type of scenarios: heterogeneous multicore, like multicore with embedded GPUs, and manycore chips like GPUs [Xie et al. 2015].

Another aspect of the memory hierarchy is the inclusion property. It goes without saying that most hierarchies now do not support inclusion, except at the last-level cache. The main reason for that is to make the best use of the chip area. With inclusion, data is replicated in all the higher cache levels (higher means closer to the processor). As the number of cores increases, the data replication increases, especially when there is a lot of data sharing among cores. But, if we do not have inclusion property at the last-level cache, which is almost one big shared cache before going off-chip, then coherence will happen off-chip which will result in very high performance degradation.

3.2 Interconnect

We are in the big data era. Machines are number crunching this huge amount of data. But in order to solve big and important problems, data has to be moved both in the memory hierarchy and among cores/processors. This makes the interconnect a vital part of overall performance in any system.

There are several aspects of interconnect in any system.

* The topology: What is the shape of interconnection among the different pieces?

A piece can be a processor, a computing node, a switch, etc.

* The granularity: What are we connecting (cores, chips, full cabinets, …)?
* The technology: What is the material we are using (fiber optics, copper, …)? A single system can use different technologies at different granularity levels. That is, using copper to connect cores, fiber optics to connect processors, and so on.
* The protocol: How are data encoded as electrical or laser signals?
* The switching and routing technology: Data may have to be routed through several hops in order to go from a point to another point, how is this done?

Figure 3.1

(a) Indirect Interconnect (b) Direct Interconnect

The technology, protocol, and switching affect the bandwidth sustained by the interconnect and hence the overall performance of data movement. The granularity is just playground of the interconnect. For example, connecting cores inside a chip [Jerger et al. 2017, Kumar et al. 2005] has different characteristics than connecting several cabinets. Inside the chip the distances are much shorter and the bandwidth is much higher but space and power are restrictive than off-chip and among cabinets. In this section we take a look at the different aspects of interconnect.

3.2.1 Topology

The topology is the shaped of interconnection. In its general form, it is a undirected graph of connected vertices. These vertices can be cores, processors, routers, blades, cabinets, or anything we want to connect. Of course connecting all vertices to each other (i.e. fully connected graph) is very expensive in terms of area and power consumption. Each connection consumes, and dissipates, power. At the same time, we want each node, a vertex in the graph, to be able to communicate with any other node. This is where topology comes in. It answers the question: what is the best and most economical interconnections among the nodes? Nodes exchange messages. To get from node A to node B a message

A interconnection network can be direct or indirect. Suppose we have N nodes to be interconnected. For direct networks, the nodes that need to be interconnected reside inside the network. Indirect network have the nodes sitting outside the network, as shown in Figure 3.1. In the case of indirect interconnect (Figure 3.1(a)), the interconnect itself is made of routers and switches, while in direct interconnect, the node itself contains the router or switch.

Figure 3.2

(a) Bus Interconnect (b) Ring Interconnect

The topology can have a regular shape (e.g. mesh, ring, torus, etc) or irregular one. How do we compare two different topologies? One of the importance measurement is the diameter. For a message to go from node A to node B, if the two nodes are not directly connected, the message has to go through several hops (called the routing distance). The diameter is the maximum number of hops between any two nodes. The lower the diameter the better because then the message will travel less distance. Figure 3.2 (a) shows a bus interconnect. The longest routing distance is two, which the distance needed to send a message from node A to node C. So, the diameter of this topology is two. If we move to the ring interconnect in Figure 3.2 (b), we can easily see that the diameter is now one, which is better. As you may have realized, the diameter grows with the number of nodes. Only a full-connected topology has a fixed diameter of one. A ring of n nodes has a diameter of floor(n/2).

Beside the ring and bus, there are many different interconnect topologies. Figure 3.3 shows some examples. As you can see from the figure, some topologies are indirect interconnect (crossbar, tree, and omega) and some are direct interconnect (hypercube, mesh, and torus). Of course there are systems with irregular topology. The interconnect can be sophisticated to include contention control through routing, in which case messages sent from source to destination may take different routes depending on which links are heavily used.

That was about the topology. How about the technology used for the links?

3.2.2 Technology

The interconnect was designed mainly using copper from the old day and till. Let’s take on-chip as an example. An link between two entities (core, cache, etc) is a group of parallel copper wires. The more parallel wires we put the higher the bandwidth we get because we can move more bits in parallel. But more parallel wires means more chip area. If we don’t have this needed area then we can make each copper thinner. Thinner wire increases its resistance and hence its temperature and the power it dissipates. So, we have a tradeoff of bandwidth vs power. This tradeoff is not the only problem we face. Wire delay is a performance bottleneck.

Figure 3.3

Few Examples of Different Topologies

Moving data around from core to shared cache or among different banks of a cache is affecting performance in a negative way. The communication is also affecting programming. Programmers now need to manage data locality (to reduce data movement) and data communication (to move data in an efficient way). If there is a broadcast channel and it has low latency then programmer’s task will be easier. This is where optical interconnection comes into play.

Using photonics, we have several advantages. The power consumption is less than copper. Contention is not a big problem because you can use different wavelengths. You can have arbitrary interconnection because two overlapping laser beams will not interfere. In [Kurian et al. 2010], for example, the authors use nanophotonic technology to implement a fast efficient global broadcast network implemented as a ring connecting all the cores on-chip. This network can be used together with a traditional point-to-point copper network. Optical interconnection is getting more and more momentum and is already in use on-board and across boards [Kobrinsky and et al. 2004, Kodi and Louri 2007, Nitta et al. 2013, Xue et al. 2010]. Figure 3.4 shows a typical optical interconnect that can be used on chip. The main processor is of course electrical (the logic part). There must be a modulator, to transform electrical pulses into laser beam by modulating a laser bean using a driver. The modulated laser beam is directed, using waveguide, to its destination where a detector returns it back to its electrical form to be consumed by the logic. The main drawback for optical interconnection is that it is still more expensive than traditional copper interconnection.

Figure 3.4

Generic Setup for On-Chip Optical Communication

As we can see, there is heterogeneity here too because we can have optical and copper interconnect in the same system. Each network can have different topology too.

Now that we can have different topologies and different technologies, how are data sent across the interconnect links? There must be some kind of protocols.

3.2.3 Protocol

In computer systems, data moves from an entity to the other. These data take the form of stream of bits, messages, packets, … (depending on how you look at it). How do these data move? There must be a protocol between sending and receiving partners, whether these partners are CPUs, cache banks, or even full-cabinets. There have been many protocols in the last several decades. In this section, we will look at three of the most widely used protocols: HyperTransport, PCI-express (PCIe), and NVLINK.

Table 3.1

Bandwidth Provided by HT

| **Version** | **Max. Aggregate Bidirectional Bandwidth (GB/s)** |
| --- | --- |
| 1.0 | 12.8 |
| 2.0 | 22.4 |
| 3.0 | 41.6 |
| 3.1 | 51.2 |

**HyperTransport (HT)**1 is point-to-point link connections CPUs among each other and connecting CPUs to I/O devices. It was first introduced by AMD, in 2001, then has undergone several updates by many partners. It is a packet-based technology. That is, messages sent are divided into packets, some packets are for control and management while the rest of the packets are for payload, and each packet is sent individually to the destination where it is combined with the other packets to reconstruct the original message. This technology is used on-board and with switches and routers but not on-chip because. Table 3.1 shows maximum aggregate bandwidth that HT can provide at its maximum frequency.

Table 3.2

Bandwidth Provided by PCIe

| **Version** | **Bandwidth/Lane/Way (GB/s)** |
| --- | --- |
| 1.x | 0.25 |
| 2.x | 0.5 |
| 3.x | 1 |
| 4.0 | 2 |
| 5.0 | 4 |

**PCIe**, Peripheral Component Interface Express, is usually found on board. It is a point-to-point serial interconnect that is packet-based. PCIe link can range from one lane (x1) to 32 lanes (x32). Each lane consists of two wires each one is unidirectional. Table 3.2 shows the the typical speed of each wire (i.e. way) in a lane. Some numbers are approximations. PCIe v4.0 is just out of the door (in 2018) and v5.0 is still under development and revision.

**NVLINK** is introduced by NVIDIA to connect GPUs to each other and to the CPU. Previously, GPUs were connectd to CPU using PCIe. But PCIe became a bottleneck for the high-bandwidth requirement of GPUs. This is why NVIDIA introduce NVLINK. NVLink uses High-Speed Signaling interconnect (NVHS) introduced by NVIDIA. This signaling transmits data over a differential pairs. Eight of these differential form a Sub-Link. Each sub-link sends data in one direction. Two sublink (each at different directions) form a Link. This link can connect a CPU to a GPU or connect two GPUs. NVILINK 1.0 has a total bandwidth of 160 GB/s while 2.0 has 300 GB/s.

Figure 3.5

Dragonfly Interconnect

3.2.4 Examples

We have seen examples of interconnect inside the chip and on-board. What about higher granularity? In this section we will take a quick look at some interconnect at high-granularity in supercomputers. The trend now is to use switches with more skinny ports, that is, ports with fewer parallel bits, than fewer fatter ports. The main reason for that is that signaling rates, the electrical signals used to send the data, have gone up relative to the sizes of the network packets. This means the penalty for sending large packets through skinny ports (i.e. serialization) gone down. Another trend is to have switches with more ports. These are called high-radix switches. The implication of this is that we can have network topology with lower diameter. Now, let’s see some examples.

**Cray Aries interconnect** has a drangonfly topology. This is a hierarchical topology introduced in 2008 [Kim et al. 2008]. At the high-level there are groups which are connected in an all-to-all links. This means each group has a direct link to each other group. The topology inside each group, the lower level, can be anything, as shown in Figure 3.5. This hierarchical organization comes in very hand in supercomputers for example where we need to have interconnects at different levels as we will see in the examples discussed in Section 3.3. The Aries router can support different interconnection technologies and protocols (PCIe, Optical, etc).

Figure 3.6

Intel Omni-Path Interconnect

**Intel Omni-Path** interconnect is built with three goals in mind: low latency, low power, and high throughput. Intel wants to use this technology for future exascale computing. So, scalability must also be a goal. This interconnect has a fabric manager chip that sees the whole picture of the topology. The manager cooperates with the switches in the interconnect to decide on the best routes for each packet based on the congestion status. The main design is shown in Figure 3.6.

3.3 Examples of Supercomputers

In the previous chapter, we looked at heterogeneity inside the chip. We saw examples of chips with heterogeneous computing nodes. This chapter goes a step higher and we look at the whole system when we connect many computing nodes. As we did in the previous chapter, we will look at examples of heterogeneous machines at the system level. Because we are discussing high-performance computers, the memory, at that high level, is distributed. The design philosophy for most of them is the same: few processors in a node (or card), several nodes in a blade, several blades in a cabinet, and then multiple cabinets. At each level, there are interconnections. Processors in a node share memory. Once we are at the blade level, we have distributed memory. The naming (e.g. rack instead of blade), and number of levels, may differ among companies but this philosophy of design allows scalability, and at different granularities, when needed (adding more cards, blades, etc). Figure 3.7 summarizes what we just said.

Figure 3.7

Design Philosophy of a High-Performance Machine

3.3.1 Cray XC50 Supercomputer

Our first example is the cray XC50. The heterogeneity is obvious. It has three type of processing nodes: ARM Cavium ThunderX2, Intel Xeon, and NVIDIA Tesla P100 (Pascal architecture) GPU. A computer node has two processors: Intel Xeon and NVIDIA P100. Every four nodes form a blade. Up to 16 blades form a chassis. A cabinet can have up to three of this chassis and a claimed peak performance of 1 PFLOS (Peta Floating-point Operations per second). In a following step, Cray introduced the ARM processor Cavium ThunderX2 in their XC50 supercomputer instead of the Intel Xeon. With this step, they made their machine very flexible to accommodate different host processors.

Each computing node has a one-to-one matching of a multicore (whether from Intel or ARM) and an accelerator. This makes for a balanced compute to accelerator. This balance, coupled with choices of CPU (Intel vs ARM), allows Cray to cater to different heterogeneous workloads. Even though the memory is distributed across cards, but the address space of general purpose processors (i.e. the nonaccelerators) is extended to access all the physical memory in the system. This makes the machine programmable with languages like MPI and also with scheme of partitioned global address spaces (PGAS) like Unifided Parallel C (UPC) and Fortran with Coarrays. This capability of allowing processors to access all the physical address of the system is enabled through the Aries interconnect.

The Aries interconnect by Cray is a chip (router), a topology, an interconnect technology, and a protocol. There is an Aries chip in each computing node. In that protocol it uses network address of three parts: (1) 18-bit node identifier (2) 12-bit memory domain handle associated with a memory segment registered at the remote node (3) 40-bit offset into this segment. This makes a 70-bit address that allows a process in a node to access all physical addresses in the system. At the technology front, the Aries interconnect uses optical connection, and for a good reason. The down side with small-diameter topologies is that the links (i.e. cables) are a bit long in length, for example 15 meters. With that length, using copper interconnect will be very slow. This is why Cray uses optical interconnect here. However, the optical interconnect is not used everywhere in that system but is used in tandem with copper wire, as we will see shortly. The topology used is called dragonfly. This topology has a very small diameter. There is an all-to-all links inside the chassis. This is done with copper wires because the length is short. Then there is another all-to-all links among six chassis. We saw earlier that every three chassis form a cabinet. So, we have an all-to-all among the chassis forming two cabinets. At that level, of chassis, the connections are done in copper too for two reasons. The first is to reduce the cost of the system because optical interconnect is very expensive. The second is that the wires are short so copper wires will not hurt performance. Among groups of two-cabinets we have optical cables and connected in an all-to-all topology because here the cable’s length is high. That is, connection between any two points does not need to go through many hops, actually only two hops if the two nodes are in the same group of two cabinets, based on the description above, and can get between any two nodes in the system with at most five hops.

3.3.2 Sunway TaihuLight Supercomputer

The Sunway TaihuLight supercomputer, developed at the National Research Center of Parallel Computer Engineering and Technology (NRCPC) in China, is the top ranked supercomputer in the Top500 list of November 2017. This machine is built around the custom designed chinese 1.45 GHz SW26010 (ShenWei) processor. Each core supports 1 thread. The simplicity of the design makes it very efficient in providing high FLOPS. 65 cores form a core group. One core of these 65 cores is used for management and the other for computations. Those computation cores are organized as 8x8 grid. Every four core groups form a node. The core groups communicate through a network-on-chip. A group of 256 nodes form supernode. Every four supernodes form a cabinet and the whole system has 40 cabinets. This means this machines have a bit over 10.6 million cores. The main design goal of the whole system is high-efficiency in floating-point operations. Its performance is 93 PFLOPS. There is an interconnect within each cabinet and another one to tie all the cabinets together. Those interconnects are custom developed but depend mostly on PCIe connections. The main draw back of that design is that the memory system is very slow. This is the price paid to get a very high GFLOPS/Watt relative to the other machines in the top 10 of the Top500 list. So, computation is great but moving data is not.

You may wonder why we are bringing this machine as an example of heterogeneous system even though it has one type of computing nodes? The heterogeneity here is very obvious in the interconnection: within a node, within supernode, in a cabinet, and across cabinets. This makes data placement and movement challenging. Even though this supercomputer was the top of the Top500 list based on the LINPACK benchmark, the golden standard of HPC for over two decades and that emphasizes on floating point muscles, it did not show great performance for the HPCG benchmark that emphasizes more on data movement.

3.3.3 Titan Supercomputer

Titan supercomputer, from Oak Ridge National Laboratory, is ranked 5th in the Top500 list. We can see the same design philosophy: processors to nodes to blades to cabinet. A node consists of a 16-core AMD Opteron processor and an NVIDIA Kepler GPU. The AMD processor is connected, through DDR3 to 32GB of memory. The GPU is connected, through GDDR5, to 6GB of memory. The CPU and GPU are connected with PCIe. All the above forms a node. Every four nodes form a blade. A cabinet has 24 blades. The system consists of 200 cabinets, making the total number of nodes 18,688. Every two nodes are connected to Cray’s Gemini interconnect. This interconnect forms a 3D torus, as shown in Figure 3.8.

Figure 3.8

Main Design of Titan

3.4 Challenges Facing Heterogeneous Computing

As computer systems become more and more heterogeneous, more challenges become even more serious.

3.4.1 Security

With the sophistication we have in modern machines in interconnect, processing nodes, memory systems, and storage keeping these machines safe in this interconnected world is a challenge. Moreover, the amount, type, and speed of data to be processed are also becoming heterogeneous, complicating the whole problem of securing the whole system from cyber-attacks.

In 2008, when an infected flash drive was inserted into one of the U.S. Department of Defense (DoD) laptops, the DoD suffered a significant compromise of its classified military computer networks [new a]. The Stuxnet attack [stu] exposed the inherent problems with computer-controlled systems in critical infrastructure and industrial process control systems. This, previously classified, incident was the most significant breach of U.S. military computers ever and marked a turning point in U.S. cyber-defense strategy. It was also a turning point in computer system research and industry to make security as important as performance and cost, not only an afterthought. The sensitive information and critical applications handled by computers makes building a trusted computer system a necessity and not a luxury. This is easier said than done due to the proliferation of methods by which a computer system can be attacked. Trusted computing is a paradigm that has emerged to address the security concerns in general purpose computing systems [TCG 2008]. There are software-oriented and hardware-oriented ways to attack a system, and also software-oriented and hardware-oriented ways to add security to computer systems and build trusted platforms. To design a trusted computing system, security has to be systematically incorporated into the various stages during the design of such systems: including system architecture, hardware implementation and software implementation [McGinn-Combs 2007].

Securing heterogeneous systems is even more challenging due to the different ways programs are executing in different accelerators and the need to move a lot of data around. Prior to execution, a program resides on the disk. When this program starts execution, it is copied to the system memory to be fetched and executed by the processor. This program can be tampered with while it is on the disk [Abualsamid 1998, Hughes and Murray 2005, Nijim et al. 2006, Ruan et al. 2009], on the bus from disk to memory or from memory to the processor [Coburn et al. 2005, Elbaz et al. 2005, Su et al. 2009], while in the memory [Di Crescenzo 2005, Vaslin et al. 2009, Yan et al. 2006], or when moving from a node to node (or blade to blade or cabinet to cabinet in large systems). or can be modified by a malicious software [Lin 2008]. Simply using encryption and decryption schemes is not enough to ensure that programs will be secure, because side channel attacks can be extremely dangerous [Standaert et al. 2009, Tiri 2007, Wang et al. 2009].

Computer systems can be compromised using software attacks and/or hardware attacks [Karri et al. 2010, Waksman and Sethumadhavan 2010]. The most obvious threat model is Time Of Check Time Of Use (TOCTOU) [Bratus et al. 2008], where there is a time interval between the time the software has been checked for integrity and the time this software is being used. During that time interval the software can be tampered with and illegitimately modified. Other type of attacks involve making use of programs vulnerabilities. Most programs usually contain vulnerabilities. Over the years, attackers have devised various attacks to exploit vulnerabilities in programs and transfer control to their attack code. Some of the most common types of attacks are:

* **Buffer overflow** [Cowan et al. 1998] attacks arise due to lack of bounds checking on the size of input array being stored in the buffer. An attacker makes use of an unchecked buffer in a program and overwrites it with his own data, thus allowing unintended control transfer to his own attack code. It is also known as the “Stack Smashing Attack” and is one of the highest reported vulnerabilities to CER2. Several hardware based solutions like a secure return address stack [Lee et al. 2003], using a non-executable stack [Cowan et al. 1998], etc. have been proposed to thwart these attacks. Secure languages such as Cyclone, which perform array bounds checking have also been proposed. As more and more applications are developed, buffer overflows attacks still form a major share of the vulnerabilities reported.
* **Return-into-libc** [Nergal 2001] exploits buffer overflow to overwrite the return address with the address of a C library function such as system(). Instead of returning into code located in the stack, control is transferred to a memory area occupied by a dynamic library. Since it uses existing code rather than the attacker’s shell code, a return-into-libc attack is very difficult to detect.
* **Code injection** [Kc et al. 2003a] attacks inject malicious code into a running application and then cause the injected code to be executed. The execution of the injected code allows the attacker to gain the privileges of the executing program. Instruction Set Randomization [Kc et al. 2003b] has been proposed to counter code-injection attacks.
* **Replay** [Wheeler 2008] attacks involve fraudulent repetition of valid data transmission by an attacker. If the destination does not detect the duplicate copies of the data being sent by the attacker, the attack is successful.

As we can see, for each type of attack there are many attempts to defend against it. This means if we want to protect our program from, say, 10 different attacks, we need to implement 10 different defenses! *A unified security system is needed to ensure program integrity against most, if not all, different threats.*

On the hardware side, attacks can result from malicious piece of hardware (called hardware Trojan) inserted into the computer system [Karri et al. 2010, Waksman and Sethumadhavan 2010], or through an external attacker who got physical possession of the system (like laptop or desktop) [Tereshkin 2010]. Collectively, hardware threats can be grouped into three categories. The first category is side-channel attacks [Karri et al. 2001, Standaert et al. 2009, Tiri 2007, Wang et al. 2009]. This type of attacks compromises the system by capturing several information about program execution, such as electromagnetic signals emitted due to computations [Fiori and Musolino 2001, Gandolfi et al. 2001], or exploiting weaknesses in caches and branch predictors [Aciiçmez 2007, Aciiçmez et al. 2010, Kong et al. 2008]. The second category requires physical access to the system. By gaining access to the system, the attacker can launch attacks against encrypted disks, or make the system boot from a compromised OS, or compromise any external device such as the ethernet card [new b]. Finally, the third category of hardware threats consists of what is called hardware Trojans. Hardware Trojan is a malicious circuitry, or malicious modification of circuitry, inserted into the computer system [Clark et al. 2009, Jin et al. 2009, Potkonjak et al. 2009, Tehranipoor and Koushanfar 2010, Wang et al. 2008]. This circuit can be inserted during any phase of the chip design and fabrication: specification phase, design phase, validation phase, physical design phase, fabrication phase, or deployment. The Trojan is typically dormant and is trigerred only after some events such as specific instructions executed, specific instruction sequence, external signals, etc. When triggered, the Trojan can disable some hardware fences,cause denial of service,leak information, or cause the system to malfunction. How do we defend against all that?

Integrity checking to detect control flow anomalies anomalies at runtime involves computing a hash value of an instruction or a basic block at compile time and comparing it with the hash value calculated at run time [Gassend et al. 2003, Gelbart et al. 2005, Schuette and Shen 1987]. At microarchitecture level, there was some work that modifies the instruction that as well as the microarchitecture to secure system against malicious code injection [Fiskiran and Lee 2004]. This technique of course has some performance effect because it controls the processor pipeline and does not commit the instructions in the basic block until integrity has been checked.

For multicore processors, Orthrus [Huang et al. 2010] employs replication of the program on multiple cores to enhance security. At run time, these replicas execute on different cores with the same input and their outputs are checked for consistency. Since it is much for difficult for an adversary to successfully compromise all the replicas without causing detectable divergence the system is more secure at the expense of less efficient hardware usage. SHIELD [Patel and Parameswaran 2008] uses a dedicated security processor that monitors the applications running on the other processors.

There are still several open questions:

* We need to ensure that the schemes used to ensure security are scalable to large number of nodes.
* We need to carefully model the effect on power and performance.
* We need to extend the system to different type of accelerators.

Ensure security of systems is one of many factors affecting performance. Bandwidth is another important factor.

3.4.2 Bandwidth

We are facing bandwidth wall at different levels: off-chip and at higher levels (among blades, cabinets, …). With large number of cores per chip and huge number of processors and computing nodes in high-performance computing machines, we expect potential performance gains, but also a severe performance bottlenecks: the expected increase of bandwidth requirements.

Software applications are becoming much more sophisticated and characterized by large memory footprints. This means an increase in the number of cache memory misses and more accesses to off-chip memory, which will, in turn, put a lot of pressure on memory ports, memory bus, and socket-pins, and severely affects overall performance. As we go from on-chip to off-chip to on-board to blades and all the way to cabinets, the bandwidth required is higher but the bandwidth given by technology is lower. For instance, on-chip bandwidth is much higher than off-chip. Solving the bandwidth problem needs different solutions at different levels. Dealing with on-chip is different than off-chip, which is different than among computing nodes, and so on because the bandwidth requirement is different and the technology used is different. Let’s see how we can approach bandwidth wall at one level: managing off-chip bandwidth. We did some experiment and we present here some insights.

3.4.2.1 Adventure With Off-Chip Bandwidth

A traditional memory system, DRAM, can be optimized for latency or bandwidth but not both. So, we have a latency optimized memory for CPU and bandwidth optimized memory for GPUs, another manifestation of heterogeneity. Most of the work related to memory systems in academia/industry focuses on managing the interconnection network, reducing power consumption, or designing efficient memory system(in terms of number of cache misses). However, off-chip bandwidth has always been thought of as a technological, rather than an architectural, problem, unlike on-chip bandwidth requirements. Any cache miss is a potential for a lot of traffic, not only for bringing the missed block but also writing back victimized (i.e. the block chosen by the cache replacement policy to be evicted from the cache) dirty blocks. This bandwidth requirement affects performance, and leads to power dissipation and consumption increase in interconnects, off-chip interface, memory controllers, and memory banks. Most of the current multicore architectures use a shared on-chip last-level-cache (LLC). Since off-chip traffic is mostly generated by this on-chip LLC, this LLC needs to be bandwidth friendly LLC with as little hardware overhead, and as little negative impact to performance as possible. Let’s assume, for the sake of this adventure but without loss of generality, that we have a multicore processor with two-level cache hierarchy. LLC is level two. Let’s also assume LLC is using least-recently-used (LRU) replacement policy. LRU is not really used in real processor but it is the basic algorithm from which other, more practical, policies have emerged.

We experimented with some3 SPLASH-2 benchmark suite [Woo et al. 1995] on a multicore chip of four cores. Figure 3.9 shows the percentage of LLC accesses for which the LRU block of the accessed set is found dirty. We can see that, on average, whenever the LLC (in our case L2 cache) is accessed, 69.5% of the time the LRU block of the accessed set is dirty. This means a high chance of generating writebacks, which leads to off-chip traffic. We aim at decreasing the amount of traffic going toward the memory without increasing the traffic coming from the memory. That is, we want to decrease the amount of data written back to off-chip memory without increasing cache misses.

Figure 3.9

Ratio of Dirty LRU blocks to the Total Number of Accesses in LLC

The traffic generated by LLC is the result of the replacement policy used in this cache. There are many excellent replacement policies available in the literature [Jaleel et al. 2008, 2015, Jeong and Dubois 2003b, Kharbutli and Solihin 2005, Qureshi et al. 2006, 2007, Sheikh and Kharbutli 2010, Yang et al. 2005]. Their main goal is to reduce read misses, but they do not consider the effect on bandwidth requirement.

LRU block is victimized. But what will happen if we try to victimize a non-LRU block? And why do we victimize non-LRU block? There is a common belief that non-LRU blocks are important blocks and getting rid of them will hurt performance. This is true for L1 cache. But is it true for LLC? Table 3.3 provides evidence that violating LRU strategy, of victimizing LRU block, at LLC does not always lead to severe performance loss. The Table shows the total execution time, in terms of number of cycles, of the whole execution of SPLASH-2 benchmarks for several non-LRU schemes normalized to the LRU scheme. LRU-M means we always victimize the M*th* block from the LRU stack. For an 8-way set associative cache, which is the one we use in this experiment, LRU-7 is the most-recently used (MRU) block. As we see from the table, even when we always victimize the MRU block, the performance loss can be as little as 6%. Of course this is application dependent and hardware dependent. But it is an insight that may help directing building future systems. This gives an idea about what to expect if we deviate from traditional LRU. This means that **LRU stack in LLC is no longer an *LRU* stack, and does not represent recency behavior for the running multithreaded application**. There are two reasons for such a surprising behavior. First LLC contains blocks that have exhausted their temporal locality at L1 (or upper level caches in case of more than two levels). Therefore, blocks at LLC have lesser temporal locality than L1, which makes the concept of *recency* at LRU stack at LLC less meaningful. The second reason is that the LRU stack at LLC is the result of interference of LRU stacks of L1 caches (each core has its own L1 caches), so if the interference is destructive, it no longer represents recency information for any single thread, nor for the whole program. Different threads have their own *important* blocks, and hence their own LRU stack. When these threads share a cache, the LRU stack of that cache is not really very useful. Thus the LRU stack is a ”mix” of LRU stacks of L1 caches which may not yield a recency order of ANY running thread. Sometimes this interference is constructive, which happens when there is a high percentage of shared blocks. Programs with such constructive interference are LRU-friendly, such as raytrace for example, and are negatively affected by victimizing non-LRU blocks.

Table 3.3

Normalized Values of Number of Cycles for Victimizing a Non-LRU Block

| **Scheme/Bench** | **Barnes** | **Cholesky** | **Fft** | **Fmm** | **Radiosity** | **Radix** | **Raytrace** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LRU | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| LRU-1 | 1 | 1.03 | 1.01 | 1 | 1.01 | 1 | 1.02 |
| LRU-2 | 1 | 1.06 | 1 | 1 | 1.01 | 1.04 | 1.06 |
| LRU-3 | 1 | 1.15 | 1.01 | 1 | 1.03 | 1.07 | 1.12 |
| LRU-4 | 1.02 | 1.23 | 1.01 | 1.03 | 1.03 | 1.09 | 1.21 |
| LRU-5 | 1.04 | 1.37 | 1.02 | 1.06 | 1.03 | 1.12 | 1.36 |
| LRU-6 | 1.13 | 1.55 | 1.04 | 1.12 | 1.05 | 1.18 | 1.63 |
| LRU-7 | 1.46 | 1.69 | 1.06 | 1.21 | 1.09 | 1.25 | 2.05 |

Another evidence that the strict order of the LRU stack is losing its importance is shown in Figure 3.10. The figure shows the stack hit ratios of the LRU stack for 1, 4, and 8 cores for some of the SPLASH-2 benchmark suite. For 1 core, the result is expected and known; the majority of the hits occur at the MRU position. But as we increase the number of cores, which in our case increases the number of threads, the MRU block becomes less important. Are these findings of any use to us? Let’s see.

Figure 3.10

LRU Stack Hits for 1, 4, and 8 Cores for 8-way Set Associativity

We need to try several things. Let’s start we a simple static technique. Suppose we have an *n*-way associative cache. The blocks are placed from position 1 to *n* with the most recently used block at position 1 and the least at position *n*. In a regular LRU, whenever a new block has to be placed in the cache, the block at position *n* (the LRU block) is evicted. In our experiment, let’s call it *Modified LRU* (MLRU), we choose to victimize a clean block between LRU and LRU–M. This saves bandwidth by retaining the dirty blocks in the cache as long as possible. If, however, all the blocks from LRU to LRU–M are dirty, the LRU block is chosen to be evicted like in the regular replacement policy. For example if M=3, the technique would look for the first nondirty block starting from LRU to LRU–3 and victimize it. There is one parameter for this scheme; M. This parameter designates how many blocks we will examine to determine the victim. M=1 is the traditional LRU, since it means that we will examine only one block, which is the LRU. M=*n* means we will look at the LRU blocks and *n* – 1 other blocks on top of it in the LRU stack. So for an 8-way set-associative cache, the maximum M is 8. M is determined at design time and is not changed afterwards.

The main advantage of that static scheme is its simplicity. Once we decide on the design parameter, M in that case, the implementation is fixed. However, determining this design parameter is usually not easy because it depends on runtime information that is not available at design time, so it relies on the designer experience and a lot of simulations. Determining M is very challenging and is application dependent. We need a dynamic scheme where M changes dynamically depending on the application behavior. We can try some schemes for dynamically changing M; *Writeback-sensitive* and *LRU-sensitive*. Each of these schemes can either be Global (same M for the whole cache) or Local (M per set). In the writeback Sensitive Scheme, when M is high, we have a higher chance of reducing traffic, but also a higher chance of affecting performance. This means that M is governed by how much traffic (writebacks) is generated, and this traffic is generated only in case of a cache miss at LLC. So in writeback-sensitive, M is incremented when there is a writeback, to reduce off-chip traffic, and is decremented when there is a cache miss with no writeback, to reduce performance loss. The new value of M will be used the next time a victim is to be chosen. In LRU-Sensitive Scheme, we check the LRU block every time the cache is accessed to see whether it became dirty. If it is, then we have potential traffic if we victimize the LRU block, so we increment M to reduce potential traffic. We decrement M whenever there is a cache miss, at LLC, to reduce potential performance loss. The technique works on the fact that if LRU is made dirty then there are more chances of a writeback, so we increase the value of M to check more blocks for a clean one. If however the application is LRU friendly it will result in misses, each miss would reduce the value of M bringing it back to 1 (LRU). Thus this scheme tries to reduce potential bandwidth. It assumes implicitly that a dirty LRU will generate bandwidth. This assumption is not correct all the time this is why the former proposed scheme (writeback-sensitive) turns out to be better. We will show results of these experiments shortly. But for now let’s continue our brainstorming for reducing off-chip bandwidth by manipulating LLC replacement policy.

In order to reduce the complexity of the hardware, compiler support can be used. Compilers are essentially free in terms of hardware requirement. We will use compiled based profiling. The compiler will use one core only to do profiling. During profiling, the compiler builds a write frequency vector (WFV). WFV is just a histogram representation of the number of writebacks at several time intervals. During profiling, the compiler keeps a count of the number of writebacks at each interval of length *X*. Then, based on that number, an entry of the WFV is incremented by one. In our experiments, we made the compiler check the number of writebacks every 10,000 cycles. We have a WFV of 32 elements, where each element represent an interval of 10. For example, if the number of writebacks in an interval is 75, it will increment the element number 7 in WFV. The above numbers are empirical based on the SPLASH-2 suite, but can be easily changed. After the profiling phase, the compiler has a WFV representing writebacks distribution for the application at hand. The following step removes noise from WFV. We define noise as any entries that are smaller than 5% of the largest value in WFV and we remove them. This noise-removal step is necessary because we want our algorithms to be based on the frequent behavior, not some individual rare events. With the noise-free WFV, we propose two algorithms; frequency-based algorithm, and weighted average algorithm.

For Frequency-Based Algorithm: Given a noise-free WFV,

* Pick the two indices for the highest and lowest numbers above the noise level
* Min = highest number of the interval represented by the lowest index
* Max = lowest number of the interval represented by the highest index

For example, if the two indices were 7 for the lowest and 20 for the highest, Min will be 69 and Max will be 199. Min and Max represent two thresholds that will be used at run-time. These two thresholds are given to the hardware. Every *X* cycles, the hardware keeps a count of the number of writebacks from LLC in that interval. This count will either fall below Min, between Min and Max, or above Max. If it is below Min, LLC will use traditional LRU for victim selection, because it means that we have low numbers of replacements of dirty blocks and we do not want to risk losing performance. If the number of writebacks is between Min and Max, we use MLRU with M=associativity/2 as indicated in the previous section. Finally, a number above Max means we have a lot of writebacks, so we use MLRU with M=associativity.

For Weighted Average Algorithm: Min and Max are used as the frequency-based, but Min is always 0, and Max is calculated as ∑i=032WFVi\*midiSUMhttp://latex.codecogs.com/gif.latex?\sum_%7bi=0%7d%5e%7b32%7d\frac%7bWFV_%7bi%7d*mid_%7bi%7d%7d%7bSUM%7d where *WFVi* is the content of element i of the WFV vector, *midi* is the mid range of element i (for example if we are talking about element 6, it spans range 50 to 59 with mid of 55), and SUM is the sum of all non-noisy entries of all WFV. So, in this algorithm we have only two regions: below Max (where we use MLRU with M=associativity/2) and above Max (where we use MLRU with M=associativity). Max can be thought of as the weighted average of the indices based on their entries. The hardware execution is similar to the frequency-based algorithm above.

One last comment on the above thresholds is that they were computed through profiling with a single core, but will be used in a multicore environment. With multicore, and hence multithreading, the traffic is expected to be higher due to the increase in misses (coherence miss), coherence traffic, etc. So before using these thresholds, we multiply them by the total number of cores.

We can combine compiler support with dynamic adaptation. The hardware will start with the above thresholds, but will adjust them dynamically based on the behavior. It will keep track of the number of writes every *X* cycles, as indicated above. The main difference is when an *event* occurs. An event is defined as a situation where half or all the sets in the cache have dirty LRUs. Whenever this event occurs, the counter that keeps track of dirty LRUs is reset if all the sets have dirty LRUs, and Max is updated as follows. If the number of writes (numWRITES) at the current time frame (which is every *X* cycles) is larger than the current Max, then the *newMax = numWRITES – delta/4*, where delta is the difference of numWRITES and currentMax. Min is adjusted in a way to make the difference between Min and Max constant. The 4 does not represent the number of cores, but it means adding to numWRITES 25% of the difference, which decreases the effect of some bursty noisy behavior. For the weighted average method, there is no Min.

After all these suggested techniques to reduce off-chip bandwidth traffic in a multicore processor, what are the results we obtained?

We compare each technique by looking at the following parameters.

**Number of Write Backs**: This is the traffic from LLC to the memory, and is our measure of success.

**Number of Read Misses**: This is from the memory to LLC. A write miss is considered a read miss followed by a write hit.

**Number of Cycles**: The total number of cycles taken by the program till completion. So, we are trying to reduce the number of writebacks (our main measure of success), while not increasing the number of read misses (negative side effect), and with minimal impact to the total number of cycles.

Our first set of experiments compares traditional LRU with MLRU for different values of M. Figure 3.11 shows the number of writebacks normalized to LRU. All of the benchmarks show a decrease in the number of writebacks. The maximum reduction in writebacks is shown by Raytrace in which the writebacks are reduced to near 0. This means that most of the writebacks for that application were for local variables which are no longer needed or for register spilled. That is, they are dead values. Raytrace and Cholesky show a decrease of 36.8% and 93.9% respectively.

Figure 3.11

Normalized Number of Writebacks for MLRU

Although we have reduced the traffic going from LLC to off-chip, we want to be sure that we have not increased the traffic coming from off-chip to memory. Figure 3.12 shows the normalized number of read misses for different values of M. For most benchmarks the increase in misses is less than 15% except for barnes which has an increase in miss of 56.9%. The high increase in misses of barnes means that it is LRU friendly, and it has high temporal locality. For FFT and Cholesky we even see a decrease in the number of misses by 16% and 14% respectively. This means that LRU is not always the best replacement policy, and it is better sometimes to *violate* LRU to gain reduction in off-chip bandwidth.

Figure 3.12

Normalized Number of Read Misses for MLRU

Figure 3.13 shows the total number of cycles normalized to the LRU scheme. For three benchmarks (FFT, Cholesky and Radix) we see an increase in performance since the number of cycles has decreased. This is because we decreased delay caused by bus contention and memory port contention beside a reduced miss penalty because the victim is overwritten not written-back. For the benchmarks that have an increase in misses we expect some performance loss. Only three benchmarks show a decrease in performance out of which two benchmarks have increase in number of cycles less than 1% except for barnes which is 7.5%. By using M=3 or M=4 we guarantee good performance. For the rest of the results we will use MLRU with M=4 as our main static technique to compare with other techniques. Also for the rest of the experiments, we will drop FMM and Radix due to their very low number of writebacks.

Figure 3.13

Speedup

Let’s now add compiler support to the static scheme. Figure 3.14 shows the normalized number of writebacks. We compare with eager-writeback [Lee et al. 2000](a compromise between write-through and writeback), DIP [Qureshi et al. 2007] (Dynamic Insertion Policy), our techniques use compiler support (frequency-based and weighted-average) and MLRU with M=4. The best three schemes for 4 cores are the weighted-average, M=4, and DIP. However, the weighted-average becomes better as we increase the number of cores to 8, which is a sign of good scalability. The frequency-based method is doing well too, but the LRU part of it (when number of writebacks is below Min) holds it a step behind the weighted-average. Similar behavior can be said on the number of misses shown in Figure 3.15. As the number of cores increases, compiler based weighted-average method and static MLRU with M=4 are doing much better. The proposed techniques are not hurting performance as indicated by Figure 3.16.

Figure 3.14

Normalized Number of Writebacks For 4 Cores (Left) and 8 Cores (Right) (Compiler Support and Static Schemes)

Figure 3.15

Normalized Number of Read Misses For 4 Cores (Left) and 8 Cores (Right)(Compiler Support and Static Schemes)

Figure 3.16

Speedup For 4 Cores (Left) and 8 Cores (Right)(Compiler Support and Static Schemes)

tt Raytrace is the only benchmarks that suffered some performance loss. This is because the interference at LLC among the threads is constructive (blocks shared by more than one core accounts for 99% of the blocks at LLC) which makes the program very sensitive to block replacement, as was indicated earlier in Table 3.3.

Figure 3.17

Normalized Number of Writebacks For 4 Cores (Left) and 8 Cores (Right)(Compiler Support and Dynamic Schemes)

Let’s now see how our dynamic techniques + compiler support behave. We experimented with the four variations of the dynamic techniques: writeback-sensitive and LRU-sensitive, each one can use local (M or global (M)). We found that the global scheme is better from a price/performance point of view. The gain we get from the local schemes does not justify the extra hardware. The writeback-sensitive is better than LRU-sensitive techniques for most of the benchmarks. So we will be using writeback-sensitive with global M as our dynamic technique (we will call it dynamic M). Figure 3.17 compares the total number of writebacks normalized to LRU. The weighted-average and dynamic M schemes are doing the best on average. This means that they are the most accurate capturing blocks behavior. The LRU component of the frequency-based scheme is affecting its ability to save on off-chip bandwidth.

Figure 3.18 compare the same techniques in terms of read misses. With the exception of Barnes, dynamic M and weighted-average methods are doing the best on average, together with DIP. To understand the behavior of barnes it is important to see the number of dirty blocks it has. Figure 3.19 shows the percentage of dirty blocks to the total number of blocks at LLC for barnes over time. There are phases where this program has 80% of its blocks dirty. This causes M to be very high which increases the likelihood of discarding an important block because high values of M affects barnes performance as indicated by Table 3.3 earlier. This also explains the speedup (and slowdown) shown in Figure 3.20.

What is the moral of all that? We can see from all the above that:

* Bandwidth is a performance bottleneck and big challenge.
* We are using very diverse set of applications with different requirements, another source of heterogeneity.
* It is very challenging to deal with bandwidth without affecting other factors like performance.
* The problem becomes more challenging as the number of cores and threads increases.

Figure 3.18

Normalized Number of Read Misses For 4 Cores (Left) and 8 Cores (Right)(Compiler Support and Dynamic Schemes)

Figure 3.19

Percentage of Dirty Blocks to Total Blocks for Barnes

Figure 3.20

Speedup For 4 Cores (Left) and 8 Cores (Right)(Compiler Support and Dynamic Schemes)

All of that was for off-chip bandwidth. Once we are off-chip we need to deal with bandwidth at higher granularities: among chips on-board, among boards in blades, among blandes in cabinets, and among cabinets.

3.4.2.2 How About Bandwidth-Wall in Other Levels?

At higher level of granularity, dealing with bandwidth depends on several factors. One factor is the implementation that consists of technology used (copper vs optical) and protocol and signaling used, as we have discussed earlier in this chapter. Another very important factor is *data locality*. As you may have guessed, it means structuring your parallel program such that processes (or threads) have the data they need near where they execute. This will reduce the need for frequent data movement and this is a big advantage by itself. Energy efficiency of wires is not improving and computation cost is much lower than communication cost.

Data locality depends mainly on the programmer. The compiler and OS can have a role but the programmer knows his/her program well and can structure it accordingly. The main issue here is that this complicates the task of the programmer and affects the productivity. Therefore data locality abstraction is a need. Data locality abstraction is a way of expressing computations so that information about proximity of data can be communicated to optimizing software stack [Unat et al. 2017]. The abstraction takes several forms: libraries, data structures, languages, or runtime. Other tools that can help in data abstraction are: debuggers, profilers, and of course compilers. If done well, and it is still an ongoing task, data locality abstraction will boost programmer’s productivity and will be a big step towards performance portability.

Compilers currently have some kind of abstraction in parallel languages. Some have local data by default (e.g. MPI), some have data global by default (e.g. PGAS languages), and some do not expose anything to the programmer (e.g. OpenMP). PGAS are easier to adopt by harder to get performance easily.

In order to have runtime support for data locality abstraction, we need to have two models: application model and architecture model. The application model consists of a set of metrics to present the application behavior, for example an affinity graph. This is still an active research topic. The architecture model encompasses computation nodes and their heterogeneity, communication cost, and memory access cost. Using these two models, the runtime can optimize data movement with minimum programmer’s effort.

3.5 In Conclusion

In this chapter we went from inside the chip to the system level. We saw how to integrate several chips to make a big machine. The design manifests itself in different forms: the type of computing nodes, the interconnection, and the memory hierarchy. In very large machines, such as supercomputers, the interconnect is usually implemented at several levels: within a chip, within nodes, within blades, within cabinets, and among cabinets. These levels of interconnect introduces a severe level of heterogeneity in data movement latency, let alone the heterogeneity in number crunching performance introduced by the different computing nodes. How to program those type of heterogeneous machines? This is the topic of the next chapter.

https://www.hypertransport.org/

http://www.cert.org defines CERT as an organization devoted to ensuring that appropriate technology and systems management practices are used to resist attacks on networked systems and to limiting damage and ensure continuity of critical services in spite of successful attacks, accidents, or failures. CERT is not an acronym.

We have chosen the benchmarks with the largest memory footprint from the SPLASH-2 suite. The input set of the benchmarks does not vary with number of cores.

Bibliography A. Abella and A. Gonzalez. June 2006. Heterogeneous way-size cache. In International Conference on Supercomputing (ICS), pp. 239–248. A. Abualsamid. June 1998. Pgp disk’s security takes a bite out of crime. Netw. Comput., 9. O. Aciiçmez. Yet another MicroArchitectural attack:. In *Proceedings of the 2007 ACM workshop on Computer security architecture - CSAW '07*, O. Aciiçmez, B. B. Brumley, and P. Grabher. 2010. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems -- CHES 2010*, Security and Cryptology, pp. 110–124. S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Feb. 2017. Compute caches. In The 23rd IEEE Symposium on High Performance Computer Architecture (HPCA). DOI: 10.1109/hpca.2017.21. A. Agarwal and et al. 1988. An evaluation of directory schemes for cache coherence. In 25 Years ISCA: Retrospectives and Reprints. A. Agarwal and et al. 2004. Evaluation the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In Proc. 32nd Int’l Symposium on Computer Architecture (ISCA). A. Agarwal and S. D. Pudar. 1993. Column-associative cache: A technique for reducing the miss rate of direct-mapped caches. In Proc. 20th International Symposium on Computer Architecture, pp. 179–190. H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. 2004. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In Proc. 42nd ACM Southeast Conference. D. H. Albonesi. 2002. Selective cache ways: On-demand cache resource allocation. Journal of Instruction-Level Parallelism. J. Archibald and J.-L. Baer. May 1986. Cache coherence protocols: Evaluation using a multiprocessor simulation model. ACM Transactions on Computer Systems. E. Azarkhish, D. Rossi, I. Loi, and L. Benini. 2016. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637, pp. 19–31. Springer-Verlag New York, Inc., New York, NY, USA. ISBN 978-3-319-30694-0. . R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. July 2014. Near-data processing: Insights from a micro-46 workshop. IEE MICRO magazine, 34(4): 36–42. ISSN 0272-1732. DOI: 10.1109/MM.2014.55. B. Beckmann and D. Wood. December 2004. Managing wire delay in large chip-multiprocessor caches. In Proc. 37th Int’l Annual Symp. on Microarchitecture ( Micro-37). M.T. Billingsley, III, B. R. Tibbitts, and A. D. George. 2010. Improving upc productivity via integrated development tools. In Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS ’10, pp. 8:1–8:9. ACM, New York, NY, USA. ISBN 978-1-4503-0461-0. . DOI: 10.1145/2020373.2020381. A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. Jan 2017. Lazypim: An efficient cache coherence mechanism for processing-in-memory. IEEE Computer Architecture Letters, 16(1): 46–50. ISSN 1556-6056. DOI: 10.1109/LCA.2016.2577557. P. Bose. Feb. 2013. Is dark silicon real?: Technical perspective. Commun. ACM, 56(2): 92–92. ISSN 0001-0782. . DOI: 10.1145/2408776.2408796. J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Nov. 2017. Emerging nvm: A survey on architectural integration and research challenges. ACM Trans. Des. Autom. Electron. Syst., 23(2): 14:1–14:32. ISSN 1084-4309. . DOI: 10.1145/3131848. R.K. Braithwaite, W.-c. Feng, and P. S. McCormick. 2012. Automatic numa characterizationusing cbench. In Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE ’12, pp. 295–298. ACM, New York, NY, USA. ISBN 978-1-4503-1202-8. . DOI: 10.1145/2188286.2188342. S. Bratus, N. D’Cunha, E. Sparks, and S. W. Smith. 2008. Toctou, traps, and trusted computing. In Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications, Trust ’08, pp. 14–32. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-540-68978-2. B. Calder, D. Grunwald, and J. Emer. 1996. Predictive sequential associative cache. In Proc. 2nd International Symposium on High Performance Computer Architecture. F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. April 2004. Productivity analysis of the upc language. In 3rd International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS), pp. 254–. DOI: 10.1109/IPDPS.2004.1303318. S. Chandrasekaran and G. Juckeland, eds. 2018. OpenACC for Programmers: Concepts and Strategies. Addison-Wesley. J. Chang and G. S. Sohi. 2006. Cooperative caching for chip multiprocessors. In Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA), pp. 264–276. L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. Carter. June 2006. Interconnect-aware coherence protocols for chip multiprocessors. In Proc. 33rd IEEE/ACM International Symposium on Computer Architecture.