Hardware Cache

Related terms:

Shared Memory, Graphics Processing Unit, Memory Access, Message Passing Interface, Message Passing, Scratchpad Memory, Cache Coherence

View all Topics

Virtual Memory

Bruce Jacob, ... David T. Wang, in Memory Systems, 2008

31.1.1 Address Spaces and the Main Memory Cache

Just as hardware caches can have many different organizations, so can the main memory cache, including a spectrum of designs from direct-mapped to fully associative. Figure 31.1 illustrates a few choices. Note that virtual memory was invented at a time when physical memory was expensive and typical systems had very little of it. A fully associative organization was chosen so that the operating system could place a virtual page into any available slot in physical memory, thus ensuring that no space would go unused (a guarantee that could not be made for a direct-mapped or set-associative organization). This design reduced contention for main memory as far as possible, but at the cost of making the act of accessing main memory more complex and perhaps more time-consuming.

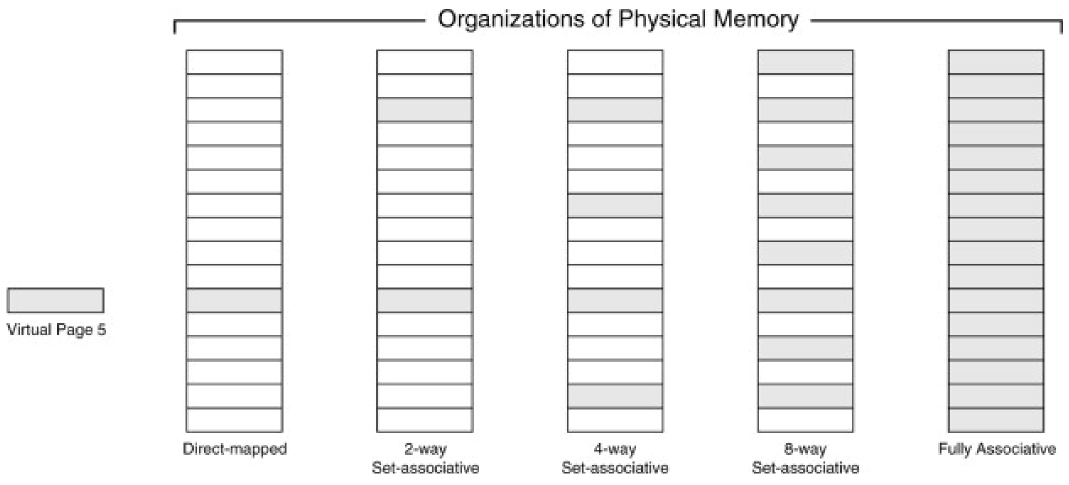

FIGURE 31.1. Associative organizations of main memory. These diagrams illustrate the placement of a virtual page (page 0x05) within aphysical memory of 16 pages. If the memory is organized as a direct-mapped cache, the page can only map to one location. If the memory is 2-way set-associative, the page can map to two locations. If the memory is 4-way set-associative, the page can map to four locations, etc. A fully associative organization allows the page to map to any location—this is the organization used most often in today's operating systems, though set-associative organizations have been suggested before to solve cache-coherence problems and to speed TLB access times.

This design decision has never been seriously challenged by later systems, and the fully associative organization is still in use. However, there have been proposals to use set-associative designs to solve specific problems. Taylor describes a hardware caching mechanism (the *TLB slice*) used in conjunction with a speculative TLB lookup to speed up the access time of the TLB [Taylor et al. 1990]. Taylor suggests that restricting the degree of set associativity when locating pages in main memory would increase the hit rate of the caching mechanism. It should be noted that if the hardware could impose a set-associative main memory organization on the operating system, the caching mechanism described in the paper would be superfluous, i.e., the speculative TLB lookup would work just as well without the TLB slice. Chiueh and Katz [1992] suggest a set-associative organization for main memory to move the TLB lookup off the critical path to the physically indexed Level-1 processor cache and to allow the cache to be larger than the page size times the cache's associativity. Similarly, the SunOS operating system, to eliminate the possibility of data corruption, aligns virtual-address aliases on boundaries at least as large as the largest virtual cache [Cheng 1987]. Kessler and Hill [1992] propose a similar mechanism to be used in conjunction with optimizing compilers so that the compiler's careful placement of code and data objects is not accidentally undermined by the virtual memory system. The consistency of virtual caches is described in more detail in Chapter 4.

> Read full chapter

Applying Software-Managed Caching and CPU/GPU Task Scheduling for Accelerating Dynamic Workloads

Mark Silberstein, ... John D. Owens, in GPU Computing Gems Jade Edition, 2012

36.1 Introduction, Problem Statement, and Context

This chapter endeavors to assist developers in overcoming two major bottlenecks of the high-end GPU platforms: memory bandwidth to the main (global) memory of the GPU, and the CPU-GPU communications. We faced both these problems when developing an application for computing the probability of evidence in probabilistic networks, and only by solving both did we achieve the desired performance improvement. Yet we believe that our techniques are applicable in a general context, and can be employed together and separately. In the chapter we describe the solution for each problem and demonstrate their combined effect on a real application as a whole.

Memory access optimization is among the main tools for improving application performance in CPUs and GPUs. It is of added importance if the algorithm has a low compute-to-memory access ratio. Often the same data are reused many times, and reorganizing the computations to exploit small but fast on-die caches might thus reduce the main memory bandwidth pressure and improve performance.

Hardware caches employ input-independent replacement algorithms, such as Least Recently Used (LRU). Maximizing cache performance to exploit data reuse requires restructuring the code so that the actual access pattern matches the cache replacement algorithm. Unfortunately, high performance is difficult and sometimes even impossible to achieve without the ability to control the replacement decisions.

Modern NVIDIA GPUs expose fast scratchpad memory shared by multiple streaming processors on a multiprocessor. By design, the scratchpad memory lacks hardware caching support;1 hence, it is the responsibility of the kernel to implement a *software-managed cache*, which implies determining which data to stage from the main memory and when to stage it. For cases where this determination is data-dependent, the decision must be made at runtime. The main challenge, then, is to minimize the overhead of the cache management code, which resides on the critical path of every memory access. In Section 36.3.1 we introduce techniques for analyzing the data access patterns and designing a read-only low-overhead software-managed cache for NVIDIA GPUs.

Kernel performance optimization, however, is only one component in making the complete application run faster. Often, despite optimizations, the kernel performance may vary substantially for different inputs. In some cases executing the kernel on a GPU may actually decrease the performance, such as when not enough parallelism is available. Furthermore, the overhead of the CPU-GPU communications over the PCI Express bus may reduce or completely cancel out the advantages of using a GPU. In Section 36.3.3 we focus on optimizing the choice of the processor for the kernel execution in applications with multiple inter-dependent kernels.

A simple approach is to greedily assign the device providing the best overall performance for a given input. It will work well for isolated kernels, where both the kernel input and output must reside on a CPU. For such cases, the data will always be transferred from the CPU to the GPU and back, thus allowing for a *local* decision that considers only the performance of a given kernel on each device.

However, for applications composed of multiple kernels with data dependencies, whereby the subsequent kernels use the results of the previous ones, different assignments or *schedules* of the computations on a CPU or a GPU may decisively influence the application running time. The schedule, which optimizes the performance of each kernel separately, is no longer sufficient for obtaining the best performance of the application as a whole.

Figure 36.1 shows a task dependency *graph* of a program for computing $A{\times}B{+}C$ for three matrices A,B,C. The nodes and edges of the graph denote kernels and their data dependencies, respectively. Computations are performed by traversing the graph according to the directionality of the edges. The computations of a node can be started only if all its predecessors in the graph are complete. In this example the first kernel computes $A{\times}B$ and the second one adds C to the result. The respective graph node labels denote the expected running time (the lower the better) of the kernel on a CPU or a GPU. Edge labels denote the data transfer times given that the adjacent nodes are executed on different devices. Input data nodes represent the original input data residing in CPU memory.

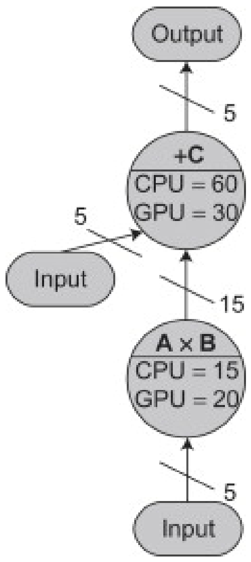

Figure 36.1. An illustration of the program task dependency graph for computing $A \times B + C$ of matrices A,B,C.

Were the schedule to consider the performance of each kernel alone, it would assign the product kernel to a CPU and the summation kernel to a GPU, yielding an execution time of 65 time units. (We assume that input transfer of matrix *C* for the summation kernel can be overlapped with the execution of the product kernel on matrices *A* and *B*.) However, the best schedule requires only 60 time units to complete, assigning both kernels to a GPU. Note that the higher cost of the data transfer between two kernels would increase the performance gap between the greedy and the optimal schedules.

We show a simple and fast algorithm which solves this scheduling problem for task dependency trees (task graphs without undirected cycles). Although the algorithm does not produce an optimal schedule (finding the optimal schedule is known to be computationally hard), it has been shown to improve the performance in real-life computations. Its main advantage is that it does not require changing the original sequential program flow, complementing other optimizations such as overlapping the data transfers with the kernel execution.

Combining the software-managed caching and GPU-CPU scheduling yields marked performance improvements over the version which does not use them. We compared the performance on random and real-life inputs using three generations of NVIDIA GPUs: GeForce 8800 GTX, GeForce GTX 285, and the Fermi-based Tesla C2050. Finally, with these techniques we obtained up to a factor of 5 speedup over the CPU-only parallel version executed on the latest dual quad-core Intel Nehalem E5540 CPUs.

> Read full chapter

Message passing interface communication protocol optimizations†

In Networks-On-Chip, 2015

10.4.2 Baseline MPI-accelerated NoC designs

This section describes the baseline communication architecture of processors. This design aims at accelerating the processing performance of MPI primitives in future massive multicore architectures by way of underlying hardware support. These multicore architectures usually present a mesh-type interconnect fabric. No hardware cache coherence exists for the first-level (L1) and second-level (L2) caches, similarly to the Intel SCC processor. A number of factors have to be considered in the performance improvement of MPI primitives through the hardware support. The basic design discussed in this chapter includes two main hardware techniques for the acceleration of MPI primitives: the NoC design and the ME [25]. Figure 10.7 shows a block diagram of the baseline implementation architecture with a 4 × 4 mesh topology. The underlying NoC design is the actual medium used to transfer messages, which can be designed with consideration for MPI communication. Each node also has an ME between the core and the network interface (NI); this ME is used to execute corresponding instructions for MPI primitives.

Figure 10.7. Block diagram of baseline communication architecture.

By directly executing the MPI primitives and interrupt service routines, the ME reduces the context switching overheads in the cores and can accelerate the software processing. The ME also performs the message buffer management for the cores as well as the fast buffer copying. This engine transfers messages to and from dynamically allocated message buffers in the memory to avoid the buffer copying between system and user buffers. This process also eliminates the need for the sending process to wait for the release of the message buffer by the communication channel. The ME also reserves a set of buffers for the incoming messages. Using the above methods, the long-message transmission protocol can be simplified, consequently reducing the transmission latency.

The ME architecture is shown in Figure 10.8. This design provides the hardware support to address the communication protocol used in the MPI implementation. Primary functionalities include serving as a middle layer between the processor core and the interface of the NoC. The ME receives the message send requests from the PE core and handles various messages from other processor cores. Two sources can trigger the ME to work: the local processor core and the NI. The local processor core may request the MPI unit perform MPI primitive functions, such that the associated communication data are transferred through this interface. Another source is the NI, which may request the ME to receive messages from the on-chip network and perform corresponding operations for handling the received messages.

Figure 10.8. Block diagram of the ME.

The ME generally comprises three key components: the preprocessing unit (PPU), parameter registers (PRs), and the MPI processing unit (MPU). The PPU is used to translate the instructions from the processor core into control signals and to generate the message passing parameters used for transfer of data that are temporarily stored in the PRs. Another important task of the PPU is to exchange data with the CPU cache for reading or writing communication data. The PRs include several registers in the ME. When MPI functions are performed, the MPI unit first receives the message parameters from the PE core and then updates these registers. The MPU is the key component of the MPI unit that performs the actual operations for MPI primitive functions, as shown in Figure 10.9a. The execution of these functions is generally performed in two separate pipelines: the send and receive pipelines. The send pipeline is used for active operations such as *MPI_Send*. The receive pipeline is used for passive operations such as *MPI_Receive*.

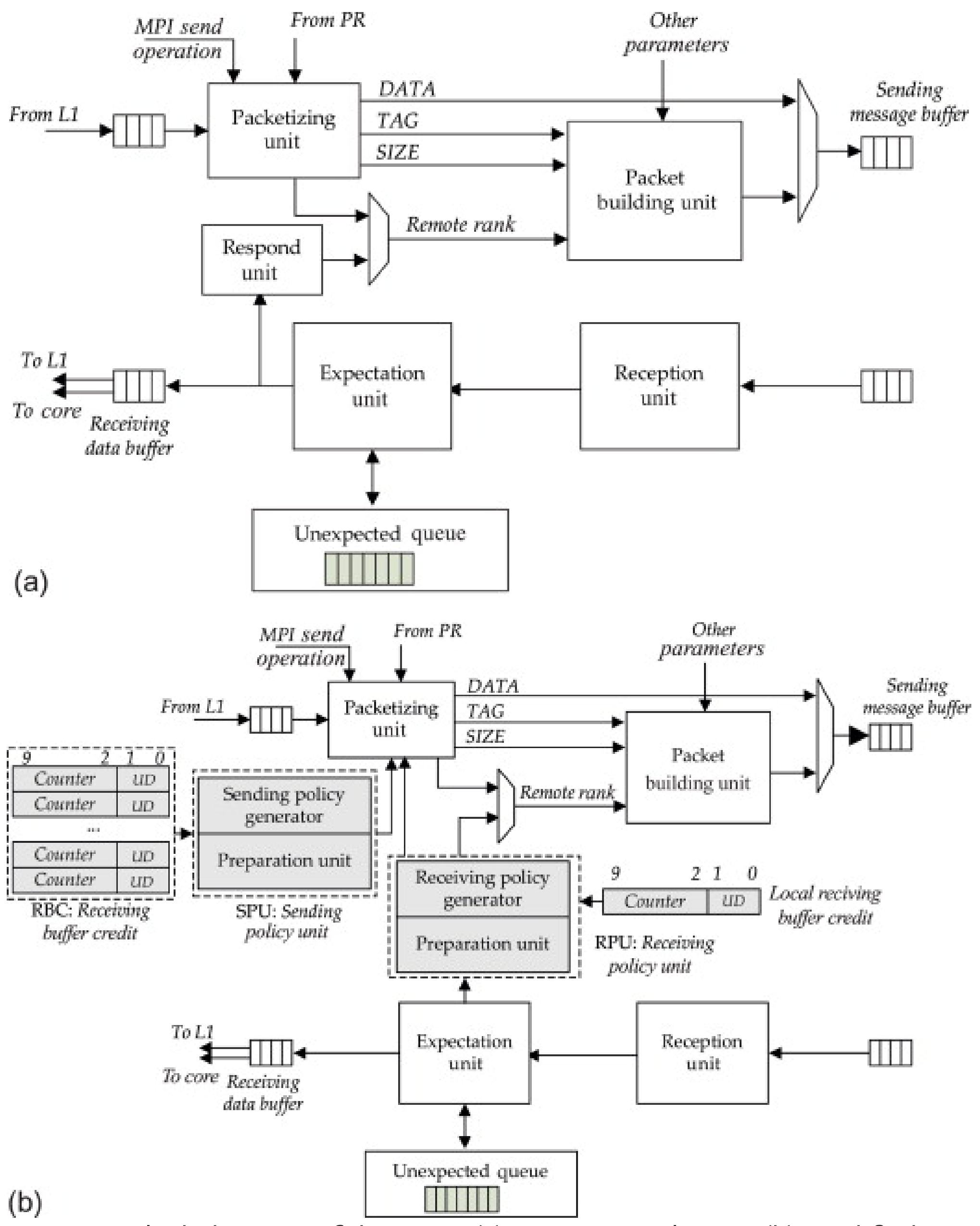

Figure 10.9. Block diagram of the MPU. (a) Conventional MPU. (b) Modified MPU for the ADCM.

> Read full chapter

An Overview of Cache Principles

Bruce Jacob, ... David T. Wang, in Memory Systems, 2008

1.2.1 Temporal Locality

Temporal locality is the tendency of programs to use data items over and again during the course of their execution. This is the founding principle behind caches and gives a clear guide to an appropriate data-management heuristic. If a program uses an instruction or data variable, it is probably a good idea to keep that instruction or data item nearby in case the program wants it again in the near future.

One corresponding data-management policy or heuristic that exploits this type of locality is *demand-fetch*. In this policy, when the program *demands* (references) an instruction or data item, the cache hardware or software *fetches* the item from memory and retains it in the cache. The policy is easily implemented in hardware. As a side effect of bringing any data (either instructions or program variables and constants) into the processor core from memory, a copy of that data is stored into the cache. Before looking to memory for a particular data item, the cache is searched for a local copy.

The only real limitation in exploiting this form of locality is cache storage size. If the cache is large enough, no data item ever need be fetched from the backing store or written to the backing store more than once. Note that the *backing store* is the term for the level of storage beneath the cache, whether that level is main memory, the disk system, or another level of cache. The term can also be used to mean, collectively, the remainder of the memory hierarchy below the cache level in question.

> Read full chapter

Multicore Embedded Systems

Colin Walls, in Embedded Software (Second Edition), 2012

10.2.2 AMP Hardware Architecture

Modern day homogeneous multicore SoCs are more complex and have many more capabilities than most older multicore designs. In the past, multicore systems were implemented on customized ASICs or FPGAs using Intellectual Property (IP) better suited to unicore environments. This IP did not have many capabilities built specifically for multicore environments, and the additional logic to support these environments was provided by the ASIC or FPGA developer, resulting in a custom hardware solution for each design.

New multicore IP such as the ARM Cortex-A9 MPCore has specific logic and capabilities to support both symmetric and asymmetric environments with vastly improved efficiency. These modern designs have multiple levels of cache that can be unified via hardware cache controllers. They also include advanced power management capabilities to maximize power and performance across multiple execution units, interrupt controllers that are focused on interrupt routing to multiple cores, and specific configuration settings to support both AMP and SMP environments.

The complexity of these designs extends beyond the cores themselves—the surrounding environment continues to evolve as well. Mesh fabrics and interconnects are replacing standard bus topologies, hardware acceleration units continue to be added for off-loading video, audio, etc, and the number of on-chip peripherals continues to grow as well. It is not uncommon to see SoC designs containing IP blocks for networking, USB, encryption/decryption, multiple busses (SPI, I2C, PCI-e), LCDs/touch panels, SD/MMC, UARTs, timers, etc. on a single chip with multiple cores.

All of these improvements have a compounding effect. New multicore IP coupled with new architecture improvements (ARMv7 architecture) and significant SoC enhancements result in complex multicore hardware designs that are well suited for both AMP and SMP environments.

> Read full chapter

Management of Cache Contents

Bruce Jacob, ... David T. Wang, in Memory Systems, 2008

3.4.4 Real Time vs. Average Case

It has long been recognized that, for good performance, applications require fast access to their data and instructions. Accordingly, general-purpose processors have offered (transparent) caches to speed up computations in general-purpose applications. Caches hold only a small fraction of a program's total data or instructions, but they are designed to retain the most important items, so that at any given moment it is likely the cache holds the desired item. However, hardware-managed caching has been found to be detrimental to real-time applications, and as a result, real-time applications often disable any hardware caches in the system.

Why is this so? The emphasis in general-purpose systems is typically speed, which is related to the *average-case* behavior of a system. In contrast, real-time designers are concerned with the accuracy and reliability of a system, which are related to the *worst-case* behavior of a system. When a real-time system is controlling critical equipment, execution time must lie within predesigned constraints, without fail. Variability in execution time is completely unacceptable when the function is a critical component, such as in the flight control system of an airplane or the antilock brake system of an automobile.

The problem with using traditional hardware-managed caches in real-time systems is that they provide a probabilistic performance boost; a cache may or may not contain the desired data at any given time. If the data is present in the cache, access is very fast. If the data is *not* present in the cache, access is very slow. Typically, the first time a memory item is requested, it is not in the cache. Further accesses to the item are likely to find the data in the cache; therefore, access will be fast. However, later memory requests to other locations might displace this item from the cache. Analysis that guarantees when a particular item will or will not be in the cache has proven difficult, so many real-time systems simply disable caching to enable schedulability analysis based on worst-case execution time.

It is difficult for software to make up for hardware's inadequacy. One solution is to pin down lines in the cache, when hardware systems support it. System software can load data and instructions into the cache and instruct the cache to disable their replacement. Another solution is to guarantee at design time the number of cache misses that will occur at run time, even if the code and data are not pinned. There have been numerous studies rearranging code and data to better fit into a cache. Examples include tiling, page coloring, cache-conscious data placement, loop interchange, unroll-and-jam, etc. [McFarling 1989, Carr 1993, Bershad et al. 1994a, Carr et al. 1994a, Calder et al. 1998]. Additionally, it has long been known that increasing set associativity decreases cache misses, particularly conflict misses. However, these mechanisms, both hardware and software, have the goal of *decreasing* the number of cache misses, not *guaranteeing* the number of cache misses. Not surprisingly, guaranteeing cache performance requires considerably more work. In fact, as we will show in the next section, the task of assigning memory addresses to code and data objects so as to eliminate cache conflicts is NP-complete.

> Read full chapter

Introduction

In Networks-On-Chip, 2015

1.5.6 The intel SCC processor

The Intel SCC processor is a prototype processor with 48 IA-32 architecture cores connected by a 6× 4 mesh network [59]. As shown in Figure 1.15, every two cores are integrated into one tile, and they are concentrated to one router. A total of four memory controllers, two on each side, are attached to two opposite sides of the mesh. The IA-32 cores are the second-generation in-order Pentium P54C cores, except that the L1 ICache/DCache capacity is upgraded from 8 to 16 kB. The L1 ICache and DCache both support four-way set associativity. They can be configured in write-through or write-back modes. Each core has a 256 kB, four-way set associative, writeback L2 cache. The L1 and L2 cache line sizes are both 32 bytes. The L2 cache is not inclusive with respect to the L1 cache. The SCC chip is implemented in 45 nm, nine-metal-layer, high-*K* CMOS technology. Its area is 567 mm2, and each tile's area is 18 mm2.

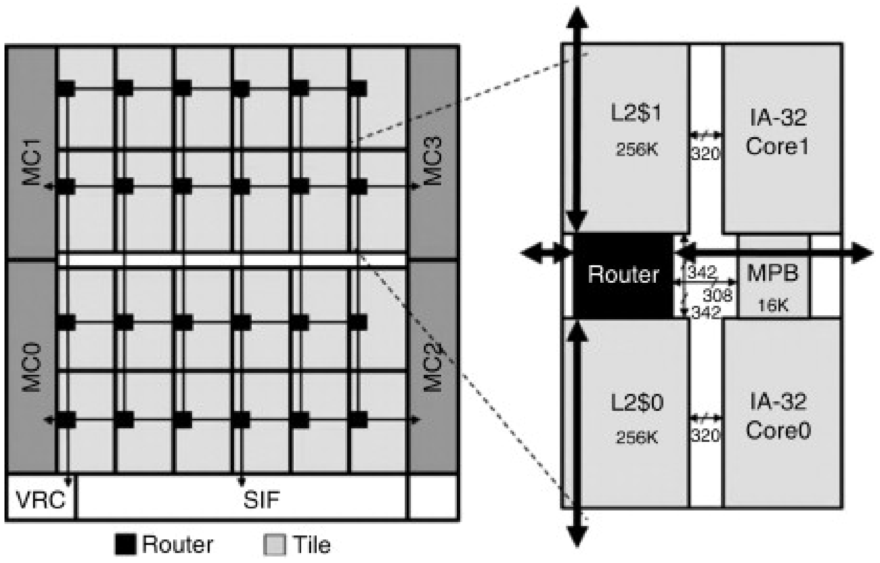

Figure 1.15. Block diagram and tile architecture of the SCC processor.From Ref. [59]. copyright 2011 IEEE.

The SCC processor does not support the hardware cache coherence protocol. Instead, it uses the message passing programming paradigm, with software-managed data consistency. To support this programming paradigm, L1 DCache lines add one message passing memory type bit to identify the line content as normal memory data or message passing data. To maintain data consistency, a core initial message write or read operation must first invalidate all message passing data cache lines. Then, the cache miss forces write or read operations to access the data in the shared memory. Each tile contains a 16 kB addressable message passing buffer to reduce the shared memory access latency.

To optimize the power consumption, the SCC chip applies dynamic voltage and frequency scaling (DVFS), which adjusts the processor's voltage and frequency according to management commands. The tile's DVFS range is between 300 MHz at 700 mV and 1.3 GHz at 1.3 V, and the NoC's range is between 60 MHz at 550 mV and 2.6 GHz at 1.3 V. The nominal usage condition is 1 GHz at 1.1 V for the tile and 2 GHz at 1.1 V for the NoC. The SCC chip consists of eight voltage islands: two voltage islands supply the mesh network and die periphery, and the remaining six voltage islands supply the processing cores, with four neighboring tiles composing one voltage island. The processor contains 28 frequency islands, with one frequency island for each tile and one frequency island for the mesh network. The remaining three frequency islands serve as the system interface, voltage regulator, and memory controllers respectively.

The mesh network of the SCC processor achieves a significant improvement over the NoC of the Teraflops processor. Its physical channel width is 144 bits, with 136 bits for payloads and 8 bits for flow control. The router architecture is optimized for such a wide link width. The router has a four-stage pipeline targeting a frequency of 2 GHz. The first stage includes the LT and buffer write, and the second stage performs the switch arbitration. The third and fourth stages are the VA and the ST respectively. To improve the performance, the router leverages the wrapped wave-front allocators. The mesh network applies lookahead *XY* DOR with VCT flow control. Each physical port configures eight VCs. To avoid protocol-level deadlock, the SCC processor employs two virtual networks: one for the request messages and the other one for the response messages. Each virtual network reserves one VC, and the remaining six VCs are shared between the two virtual networks.

> Read full chapter

Perf

In Power and Performance, 2015

8.1.2 Selecting an Event

Events are selected by populating the perf_event_attr structure and then passing it to the kernel. This data structure, as expected by the kernel ABI, is defined in /usr/include/linux/perf_event.h. This header file also includes the enumerations for the various events, as well as any other ABI definitions.

Due to the flexibility of the interface, the event attribute structure is fairly complex. Determining which fields to populate in this structure depends on the desired information.

The first step is to select the event and event type, corresponding to the perf_event_attr.config and perf_event_attr.type fields, respectively. At the time of this writing, the following predefined event types are listed in the perf_event.h header file:

PERF_TYPE_HARDWARE Hardware Event
PERF_TYPE_SOFTWARE Software Event
PERF_TYPE_TRACEPOINT Tracing Event
PERF_TYPE_HW_CACHE Hardware Cache Event

PERF_TYPE_RAW Raw Hardware Event
PERF_TYPE_BREAKPOINT Hardware Breakpoint

Additionally, the supported events and event types can be enumerated via sysfs.

The event types can be iterated via the event_source bus in sysfs. Each directory in /sys/bus/event_source/devices/ represents an event type. To select that event type, set the perf_event_attr.type field to the value in that corresponding directory's type file. Notice that the values exported via sysfs should correspond with the predefined values. For instance:

```
$ l s /sys/bus/event_source/devices /breakpoint@ cpu@ power@-
software@ tracepoint@$ cat /sys/bus/event_source/devices/software/type1$ grep
PERF_TYPE_SOFTWARE /usr/include/linux/perf_event.h  PERF_TYPE_SOFTWARE
= 1,$ cat /sys/bus/event_source/devices/power/type6
```

Each of these event types has a corresponding directory in /sys/devices. Within some of these directories is an events subdirectory. Within this events directory is a file for each predefined event. These predefined event files contain the necessary information to program the event. For example:

```
$ ls /sys/devices/cpu/eventsBranch – instructions cache– misses instructions-
Ref–cycles  branch–misses cache–referencesMem–loads stalled–cycles–back-
end bus–cyclescpu–cycles mem–stores  stalled–cycles–frontend$ cat /sys/de-
vices/cpu/events/cpu–cyclesevent=0x3c$ cat /sys/devices/cpu/events/ref–cycles-
event=0x00,umask=0x03$ls /sys/devices/power/eventsenergy–cores energy–gpu en-
ergy–pkgenergy–cores.scale energy–gpu.scale energy–pkg.scaleenergy–cores.unit-
energy–gpu.unit energy–pkg.unit
```

Notice that in the above sysfs example, an event type named "power" was available; however, this event type was not present in the prior list of ABI event types. This is because, at the time of this writing, the ABI header doesn't have a corresponding entry in the perf_type_id enum for this type. The "power" event type, introduced less than a year ago, adds events that utilize the RAPL interface, described in Section 3.2.4, to provide power consumption information. This illustrates how quickly the infrastructure is evolving. Andi Kleen has written a library, libjevents, to parse the events listed in sysfs. It is available at https://github.com/andikleen/pmu-tools/tree/master/jevents.

PERF_TYPE_HARDWARE

When perf_type_attr.type is set to PERF_TYPE_HARDWARE, the contents of perf_event_attr.config are interpreted as one of the predefined processor events. Some of these events and their meanings are listed in Table 8.2. The exact PMU counter corresponding to each of these events can be determined by searching for the event name in the ${LINUX_SRC}/arch/x86/kernel/cpu/ directory. This information can then be cross-referenced with Volume 3, Chapter 19 of the Intel® *Software Developer Manual* to determine the precise meaning.

Table 8.2. Architectural Events of Type PERF_TYPE_HARDWARE (Kleen et al., 2014)

Event	**Meaning**
PERF_COUNT_HW_CPU_CYCLES	Unhalted Core Cycles. This counter only increments when the core is in C0 and is not halted. Affected by changes in clock frequency.
PERF_COUNT_HW_INSTRUCTIONS	Instructions Retired. This counter increments once when the last μop of an instruction is retired. Only incremented once for an instruction with a REP prefix.
PERF_COUNT_HW_CACHE_REFERENCES	LLC Reference.
PERF_COUNT_HW_CACHE_MISSES	LLC Miss.
PERF_COUNT_HW_BRANCH_INSTRUCTIONS	Branch Instruction At Retirement. Due to PMU skid, the flagged instruction during sampling is typically the first instruction of the taken branch.
PERF_COUNT_HW_BRANCH_MISSES	Mispredicted Branch Instruction at Retirement. Due to PMU skid, the flagged instruction during sampling is typically the first instruction of the taken branch.
PERF_COUNT_HW_BUS_CYCLES	Unhalted Reference Cycles. Unlike Unhalted Core Cycles, this counter increments at a fixed frequency.
PERF_COUNT_HW_REF_CPU_CYCLES	This event currently uses the Fixed-Function events rather than the programmable events. Similar to BUS_CYCLES, this counter increments at a fixed rate, irrespective of frequency. Since the event code in the kernel source assumes programmable events, the event code and umask hard coded in the config are bogus.

As mentioned in the introduction, abstracting PMU events is challenging. The vast majority of events are nonarchitectural, that is, their behavior can vary from one processor architecture to the next. As a result, defining predefined events for every available event would clutter the API with thousands of events that are only useful on a specific platform. Thus, most PMU events are only accessible with the PERF_TYPE_RAW event type.

Another challenge with the predefined events is that there is no guarantee that these events will continue measuring the same counters in future kernel versions.

In summary, the predefined events of type PERF_TYPE_HARDWARE are convenient for basic PMU usage. For more serious performance work, it is better to use the PERF_TYPE_RAW events and specify the desired counters directly.

PERF_TYPE_RAW

The interpretation of perf_event_attr.config when perf_event_attr.type is set to PERF_TYPE_RAW depends on the architecture. For Intel® platforms, the expected format is that of the programmable PMU MSR, shown in Figure 6.1. Of these bits, only the ones that identify the event are required. The kernel automatically sets the other bits based on the other attributes in the perf_event_attr struct. For example, whether the OS and USR fields are set depends on the values of perf_event_attr.exclude_kernel and perf_event_attr.exclude_user. In other words, only the umask and event number fields need to be set for most events. Some events may also require setting the cmask, inverse, or edge detect fields. The encoding information for each event is listed in Volume 3, Chapter 19 of the Intel *Software Developer Manual*.

Additionally, some PMU events also require configuring other MSRs. Within the kernel are lists, partitioned by processor generation, of these events and the MSRs they utilize. When one of these events is selected, the contents of perf_event_attr.config1 and perf_event_attr.config2 are loaded into the extra MSRs. Since these extra registers vary depending on the event, the format will be specific to the event. Obviously for this to work, the event must be properly configured within the kernel's PMU lists. These lists, arrays of type struct extra_reg, can be found in ${LINUX_SRC}/arch/x86/kernel/cpu/perf_event_intel.c.

PERF_TYPE_SOFTWARE

When perf_type_attr.type is set to PERF_TYPE_SOFTWARE, the contents of perf_event_attr.config are interpreted as one of the predefined software events. Some of these events can be found in Table 8.3. As the name implies, software events are provided and updated by the kernel. Therefore, these events focus on aspects of the software's interaction with the operating system. Unlike the hardware events, these events should be available and consistent across all platforms.

Table 8.3. PERF_TYPE_SOFTWARE Events (Molnar et al., 2014; Torvalds and et al., 2014)

Event	**Meaning**
PERF_COUNT_SW_CPU_CLOCK	The wall time, as measured by a monotonic high-resolution timer.
PERF_COUNT_SW_TASK_CLOCK	The process time, as measured by a monotonic high-resolution timer.
PERF_COUNT_SW_PAGE_FAULTS	The number of page faults.
PERF_COUNT_SW_CONTEXT_SWITCHES	The number of context switches.
PERF_COUNT_SW_CPU_MIGRATIONS	The number of migrations, that is, where the process moved from one logical processor to another.
PERF_COUNT_SW_PAGE_FAULTS_MIN	The number of minor page faults, that is, where the page was present in the page cache, and therefore the fault avoided loading it from storage.
PERF_COUNT_SW_PAGE_FAULTS_MAJ	The number of major page faults, that is, where the page was not present in the page cache, and had to be fetched from storage.

PERF_EVENT_TRACEPOINT

In order to count kernel tracepoints, set perf_event_attr.type to PERF_TYPE_TRACEPOINT and set perf_event_attr.config to the tracepoint id. This value can be retrieved from debugfs by looking at the respective id file under the appropriate subdirectory in /sys/kernel/debug/tracing/events/. For more information about kernel tracepoints, see Chapter 9.

PERF_EVENT_HW_CACHE

In order to utilize hardware counters to measure cache events, the value of perf_event_attr.type is set to PERF_TYPE_HW_CACHE, and the value of perf_event_attr.config is based on Figure 8.1.

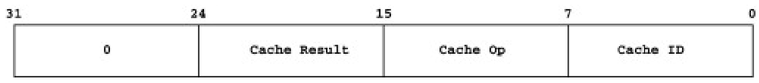

Figure 8.1. perf_event_attr.config format for PERF_TYPE_HW_CACHE.

The Cache_ID field in Figure 8.1 can be one of the following:

PERF_COUNT_HW_CACHE_L1D L1 Data Cache
PERF_COUNT_HW_CACHE_L1I L1 Instruction Cache
PERF_COUNT_HW_CACHE_LL LLC
PERF_COUNT_HW_CACHE_DTLB Data TLB
PERF_COUNT_HW_CACHE_ITLB Instruction TLB
PERF_COUNT_HW_CACHE_BPU Branch Prediction Unit
PERF_COUNT_HW_CACHE_NODE Local memory

The Cache_Op field in Figure 8.1 can be one of the following:

PERF_COUNT_HW_CACHE_OP_READ Read Access
PERF_COUNT_HW_CACHE_OP_WRITE Write Access
PERF_COUNT_HW_CACHE_OP_PREFETCH Prefetch Access

Finally, the Cache_Result field in Figure 8.1 can be one of the following:

PERF_COUNT_HW_CACHE_RESULT_ACCESS Cache Reference
PERF_COUNT_HW_CACHE_RESULT_MISS Cache Miss

PERF_TYPE_BREAKPOINT

Unlike the other event types, PERF_TYPE_BREAKPOINT doesn't use the value in perf_event_attr.config. Instead, it should be set to zero. The breakpoint is set by setting perf_event_attr.bp_type to one or more of the memory access types described in Table 8.4. Then, set perf_event_attr.bp_addr and perf_event_attr.bp_len, to set the address, as well as the length of bytes after perf_event_attr.bp_addr to break upon.

Table 8.4. Values for perf_event_attr.bp_type (Howells, 2012)

Value	Meaning
HW_BREAKPOINT_EMPTY	Don't Break
HW_BREAKPOINT_R	Break on Read
HW_BREAKPOINT_W	Break on Write
HW_BREAKPOINT_RW	Break on Reads or Write
HW_BREAKPOINT_X	Break on Code Fetch

> Read full chapter

Storage

Clint Huffman, in Windows Performance Analysis Field Guide, 2015

Hardware and terminology

This is the terminology that I use in this book. This is important because many of these terms can be subjective depending on the hardware and operating systems that you and/or your vendors work with.

I/O request packet

First, I/O stands for the input or output of data to or from a device—it is how a computer communicates with the hardware and other logical devices. An IRP is the request structure that Windows and Windows Server use to transfer data to and from devices, but this structure is used for all kinds of I/O such as network and disk. This means that some of the operating system performance counters lump network I/O and disk I/O together, making it difficult to determine what kind of I/O we are working with. I will mention some of these counters later in this chapter when analyzing disk performance with Performance Monitor.

I/Os per second

This is an industry term describing the number of I/O requests that are fulfilled per second. This term is often used to describe the amount of throughput that a disk or hardware controller can handle, but the measurement doesn't always include other contributing factors such as the size of the I/O and if it is sequential or random I/O, all of which can dramatically affect the number of I/O operations per second (IOPS) the disk can do. Therefore, it is important to specify these conditions when reporting the IOPS of a device. I might say my 7200 RPM disk drive is capable of 90 IOPS when doing 50% random reads, 50% random writes, and I/O sizes of 64 KB, or it can do 260 IOPS when doing sequential reads and writes at the same I/O size.

Disk Transfers/sec is the number of read and write operations completed per second—this is the Windows and Windows Server equivalent of IOPS, but not necessarily the number of IOPS that the hardware is doing. Windows Server could be doing 100 write operations per second, but the hardware RAID1 controller would have to do 200 IOPS (100 I/O operations to two disk drives). The counters **\LogicalDisk(*)\Disk Transfers/sec** and **\PhysicalDisk(*)\Disk Transfers/sec** measure the number of disk transfers per second to the respective logical disk or physical disk.

Hard disk drive (spindle)

The ye olde spindle is the spinning magnetic disk still widely used today for disk storage. Most spindles can only handle one I/O at a time. Now, one might think if a disk queue has more than one outstanding I/O on average, then it is overwhelmed; well, that "could" be true, but keep in mind that there are many other considerations like hardware cache built into the disk drive and controller that help dramatically with performance. Even the driver can help the drive to optimize the head position and rotation to read or write many I/Os in a single rotation.

Keep in mind that the performance of a single disk drive will vary greatly depending on the peripheral bus, hardware controller, cache, and revolutions per minute (RPM).

Solid-state drive

A solid-state drive (SSD) is a data storage device that uses solid-state memory to persist data in the same way as a hard disk drive—meaning the data is retained when power is removed. These drives are based on similar media as USB "thumb" drives, usually use less power than hard disk drives, and often have significantly faster access times than hard disk drives, making them popular in laptops and other mobile devices. There are many types of SSDs, but my intent is to cover them only from a generalized perspective.

The only disadvantage of SSDs that I know of is that it oxidizes the media when it is written to limit the number of times that the media can be written to before it is unusable. This raises concerns about its long-term viability in enterprise environments. In addition, it is currently more expensive than hard disk drives per gigabyte. With that said, I have been using SSDs under constant use for over 6 years now with no problems. As a matter of fact, so far all of my SSDs have out lived my 1 TB spindle hard disk drives bought around the same time.

Another observation that I noticed about the SSD drive in my laptop is that when there is little demand (few, if any, outstanding I/O requests in the queue), then the response times are often higher than 15 ms. This is most likely due to the drive going into a power saving mode. But when under constant load, the SSD runs extremely fast—less than 2 ms on average. This is similar to behavior that I've seen on enterprise-class servers connected to a storage area network (SAN). My point is that when analyzing disk performance, a major consideration is whether or not the disk has constant load or not. This topic is covered in the "Understanding the disk queue" section later in this chapter.

Direct-attached storage

Hard drives that are represented to the operating system as a physical disk or logical unit number (LUN) are considered direct-attached storage or DAS. The computer assumes it is a single storage device "directly attached" to the computer. With that said, hardware disk controllers can control many hard drives and represent them as one or more "physical drives" or LUNs to the computer. This means that from a Windows and Windows Server perspective, we really don't know how many physical spindles are really behind any given LUN and we don't know the physical distance to it. With that said, as of Windows 8 and Windows Server 2012 forward, if the system has the SMI-S provider or SMP provider, then it is possible to determine the physical topology of a LUN using Powershell. For more information on SMI-S, see "Getting started with SMI-S on Windows Server 2012" at http://blogs.technet.com/b/filecab/archive/2012/07/06/3507632.aspx.

In any case, my point is that if a disk controller represents a "physical disk" to Windows or Windows Server, then the disks from that hardware are considered "direct-attached."

Just a bunch of disks and RAID types

A just a bunch of disks (JBOD) is an industry term that refers to a physical grouping of disk drives that are not part of a network of drives such as a SAN or network-attached storage (NAS) device. They are simply "JBOD" where every physical disk drive is presented to the operating system (Figure 3.1).

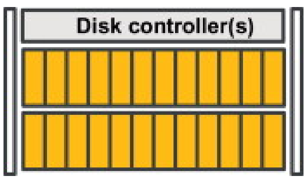

Figure 3.1. A diagram of a JBOD.

Network-attached storage

A NAS is a physical device that emulates a network file system on a TCP/IP network. A NAS can emulate the Server Message Block (SMB) protocol, which is a common protocol that Windows and Windows Server use for network file storage (Figure 3.2).

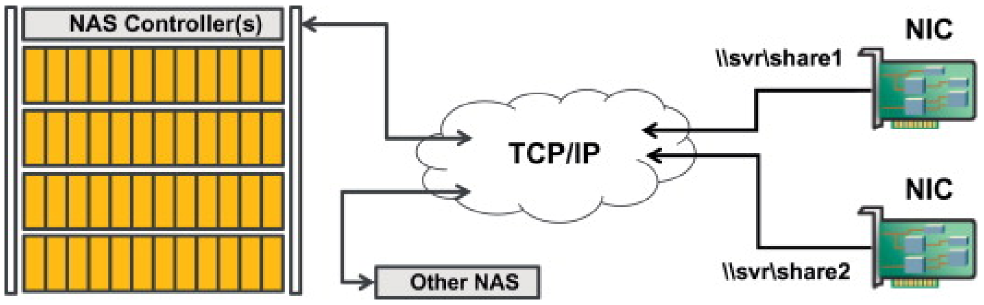

Figure 3.2. A network diagram of a NAS device and interface cards.

In regard to performance, a NAS is dependent on the performance of the network interface card (NIC) and the TCP/IP network nodes along the way. Therefore, refer to Chapter 9 "Network" on performance and troubleshooting of network resources.

SAN and Fibre Channel

A SAN is a network of storage devices (such as disk drives or SSDs) and is managed by one or more controllers. The word "network" in the name refers to the network of drives within the enclosure. A SAN can be attached to a computer system through Internet small computer systems interface (iSCSI) or Fibre Channel (FC). FC (also known as the "fabric") is made up of fiber-optic cables and optionally fiber-optic switches. A host bus adapter (HBA) is a physical interface that is installed on computers to access the FC network. The HBA allows the computer to access one or more SANs. The LUNs (a physical disk drive) presented to the operating system are considered "DAS" even though the SAN might be physically located elsewhere (Figure 3.3).

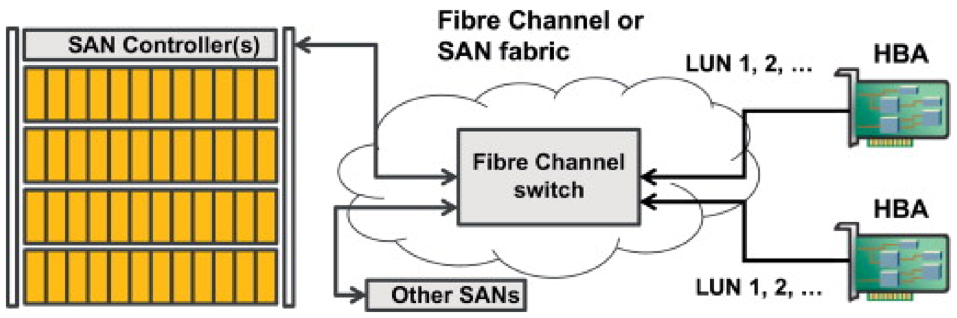

Figure 3.3. A diagram of a Fibre Channel network.

Regarding performance, SANs are often powerful and fast. With that said, in my experience, they are relatively expensive and therefore often become overused and abused. Even the fastest SAN can be overrun with enough I/O demand, so it's important to constantly monitor their performance. Unfortunately, the complexity of this type of environment often makes it difficult to understand where to start.

From my perspective, I start by comparing the SAN-attached LUNs to the performance of a single disk drive—meaning if the SAN cannot beat the performance of my 7200 USB disk drive, then there is something wrong. To be specific, I look for an average disk queue length of at least two to indicate a constant load on the disk and response times greater than 15 ms when working with 64 KB or smaller I/O sizes.

Internet small computer systems interface

iSCSI connects LUNs from a SAN to a computer just like an FC network but uses commodity NICs and a TCP/IP network. In many cases, this solution is more cost-effective because the I/O can be transferred using relatively inexpensive twisted pair "copper" cables instead of fiber-optic cables.

In regard to performance analysis, I treat these with the same initial indicators and thresholds as SAN-attached LUNs—meaning I look for constant I/O requests (disk queuing), high disk latency, and the I/O sizes (Figure 3.4).

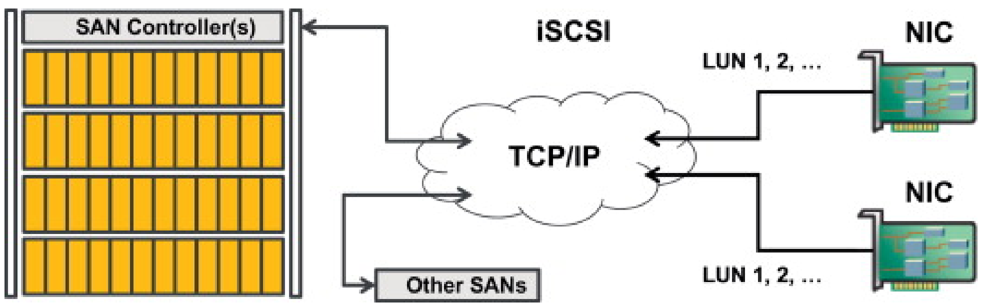

Figure 3.4. A diagram of an iSCSI network.

> Read full chapter

A comprehensive survey on Green ICT with 5G-NB-IoT: Towards sustainable planet

Sakshi Popli, ... Sanjeev Jain, in Computer Networks, 2021

2.2.1.1 Cache policies

Presently, it is discerned that a sizeable portion of multimedia traffic is produced due to duplicate downloads of popular content, numerous times. Thus a shift from conventional Tx/Rx communication to content dissemination is required. Although it has been observed in certain studies that caching at BS will reduce the Tx(transmission) energy, but additional energy will be required for caching too. However, if caching is done efficiently, it would decrease the energy requirements at BS.

Authors in [81] studied the impact of caching on BS energy efficiency. It was observed that if popular content is exploited and power cache hardware is used at BS, then EE can be improved. Besides this, they also stated that interference must also be reduced or removed for better performance (discussed further in the paper). Authors in [82] proposed a clustered content sharing technique that significantly reduces the duplicate traffic on the backhaul and as a result reduces the power consumption. In this only the required content is connected to a common edge cloud. Besides this RRH association and resource allocation are also optimized. Authors in [83] proposed an SRC (space reserved cooperative) caching scheme for Industrial-IoT in the 5G Het-Net. In this base station, cache space is distributed into 2 parts. Where one part is used to store the previously fetched data and the other part stores the provisionally buffered data in the transmission queue as the device request. Further, the authors in [84] explored a proactive caching scheme that used SVM and user social ties for the placement of cache data. Simulation results showed that the cache hit ratio improved by 14%-28% and cache operation reduced by 10%-50%. Authors in [85] suggested two algorithms to reduce BS energy consumption while ensuring QoS as per user requirement. Authors suggested that irrespective of video popularity the fraction of content must be placed at the base layer and the fraction of the left content must be placed at the enhanced layer, with condition that the fraction at the enhanced layer can't be decoded without a base layer. Hence able to achieve uniform distribution of content in a network.

From the above discussion, it is clear that the choice of content placement plays a crucial role in caching techniques. In addition to this content distribution also has a great significance that must be considered to ensure reliability while optimizing energy consumption. Besides this, the learning algorithm can also play a significant role in determining the content popularity in a zone of the network.

> Read full chapter