Chapter 2
Multi-core and Many-core Processor Architectures

Abstract No book on programming would be complete without an overview of the hardware on which the software will execute. In this chapter we outline the main design principles and solutions applied when designing these chips, as well as the challenges facing the hardware industry, together with an outlook of promising technologies not yet in common practice. This chapter's main goal is to introduce the reader to the most important processor architecture concepts (core organization, interconnects, memory architectures, support for parallel programming etc) relevant in the context of multi-core processors as well the most common processor architectures available today. We also analyze the challenges faced by processor designs as the number of cores will continue scaling and the emerging technologies—such as transactional memory, support for speculative threading, novel interconnects, 3D stacking of memory etc—that will allow continued scaling of processors in terms of available computational power.

2.1 Overview

In 2001 the first general-purpose processor that featured multiple processing cores on the same CMOS die was released: the POWER4 processor of IBM. Since then, multi-core processors have become the norm and currently the only way to improve the performance for high-end processors is to add support for more threads, either in the number of cores, or through multithreading on cores to mask long-latency operations.

There are multiple reasons why the clock rate gains of the past cannot anymore be continued. The unsustainable level of power consumption implied by higher clock rates is just the most obvious and stringent reason; equally important is the fact that wire delays rather than transistor switching will be the dominant issue for each clock cycle.

With Contribution by Mats Brorsson

A. Vajda, *Programming Many-Core Chips,*
DOI 10.1007/978-1-4419-9739-5_2, © Springer Science+Business Media, LLC 2011

Compared to single-threaded processors, multi-core processors are highly diverse and the design space is enormous. In this chapter we present the architectural principles of multi-core chip architectures, discuss some current examples and point out the critical issues with respect to scalability.

2.2 Architectural Principles

Many of the technologies behind current multi-core architectures were developed during 1975–2000 for parallel supercomputers. For instance, directory-based cache coherence was published in 1978 by Censier and Featrier ([1]) who then referred to snoopy cache coherence (although it was not called so) as "the classical solution". The development of semiconductor technology has permitted the integration of these ideas onto one single-chip multiprocessor, most often called a multi-core processor. However, as always, the trade-offs become radically different when integrated onto the same silicon die.

In addition, recent advances in semiconductor technology have permitted greater control and flexibility through separate voltage and frequency islands which provide system software the possibility to control performance and adjust power consumption to the current needs.

This section is divided into two parts: *established concepts*, which focuses on the architectural ideas forming the basis of all multi-core architectures, one way or the other, and *emerging concepts* describing ideas that have only been tried experimentally or implemented to a limited extent in commercial processors.

2.2.1 Established Concepts

The very concept of multi-core architecture implies at least three aspects that will be the main topics of this chapter:

- There are multiple computational cores
- There is a way by which these cores communicate
- The processor cores have to communicate with the outside world.

2.2.1.1 Multiple Cores

The concept of multiple cores may seem trivial at first instance. However, as we will see in the section about scalability issues there are numerous tradeoffs to consider. First of all we need to consider whether the processor should be homogeneous or expose some heterogeneity. Most current general-purpose multi-core processors are homogeneous both in instruction set architecture and performance. This means

that the cores can execute the same binaries and that it does not really matter, from functional point of view, on which core a program runs. Recent multi-core architectures, however, allow for system software to control the clock frequency for each core individually in order to either save power or to temporarily boost single-thread performance.

Most of these homogeneous architectures also implement a shared global address space with full cache coherency (which we discuss later), so that from a software perspective, one cannot distinguish one core from the other even if the process (or thread) migrates during run-time.

By contrast, a heterogeneous architecture features at least two different kinds of cores that may differ in both the instruction set architecture (ISA) and functionality and performance. The most widespread example of a heterogeneous multi-core architecture is the Cell BE architecture, jointly developed by IBM, Sony and Toshiba [2] and used in areas such as gaming devices and computers targeting high performance computing.

A homogeneous architecture with shared global memory is undoubtedly easier to program for parallelism—that is when a program can make use of the whole core—than a heterogeneous architecture where the cores do not have the same instruction set. On the other hand, in the case of an application which naturally lends itself to be partitioned into long-lived threads of control with little or regular communication it makes sense to manually put the partitions onto cores that are specialized for that specific task.

Internally the organization of the cores can show major differences. All modern core designs are today *pipelined*, where instructions are decoded and executed in stages in order to improve on overall throughput, although instruction latency is the same or even increased. Most high-performance designs also have cores with speculative dynamic instruction scheduling done in hardware. These techniques increase the average number of instructions executed per clock cycle but because of limited instruction-level parallelism (ILP) in legacy applications and since these techniques tend to be both complicated and power-hungry as well as taking up valuable silicon real estate, they are of less importance with modern multi-core architectures. In fact, with some new architectures such as Intel's Knights Corner [3], the designers have reverted to simple single-issue in-order cores (although in the Knights Corner case augmented with powerful vector instructions, in order to reduce the silicon footprint and power consumption of each core).

A counter measure for limited instruction level parallelism added to the most advanced cores is simultaneous multithreading (SMT), perhaps better known by its Intel brand name *hyper-threading*. This is a hardware technique that allows better utilization of hardware resources where a multi-issue pipeline can select instructions from two or more threads. The benefits are that for applications with abundant ILP, single-thread performance is high, while with reduced ILP, thread-level parallelism can be utilized. Simultaneous multithreading has the nice property that it is relatively cheap in terms of area (the state for each thread and a simple thread selection mechanism) and extra power consumption.

2.2.1.2 Interconnection Networks

Having multiple cores on a chip requires inter-core communication mechanisms. The historical way that the individual processors in a shared memory multiprocessor communicate has been through a *common bus* shared by all processors. This is indeed also the case in the early multi-core processors from general purpose vendors such as AMD or Intel. In order not to flood the bus with memory and I/O traffic, there are local cache memories, typically one or two levels, between the processor and the bus. The shared bus also facilitates the implementation of cache coherency (more about that in the section about shared memory support below) as it is a broadcast communication medium which means that there is always a global order of shared memory operations.

More recent designs are based on the realization that shared communication mediums such as buses are problematic both in latency and bandwidth. A shared bus has long electrical wires and if there are several potential slave units—in a multi-core processor all cores and memory sub-system are both slaves and masters—the capacitive load on the bus makes it even slower. Furthermore, the fact that several units share the bus will fundamentally limit the bandwidth from each core.

Some popular general purpose multi-core processors at the time of this writing use a cross-bar interconnect between processor modules—which include one or two levels of cache memories—and the rest which includes the last-level cache and memory interfaces. Other technologies—such as multiple ring busses, switched on-chip networks—are emerging and gaining traction due to either lower power consumption, higher bandwidth or both. As the number of cores will increase, on-chip communication networks will increasingly face scalability and power constraints.

Figure 2.1 provides an overview of the most common designs of on-chip interconnect used in today's systems.

2.2.1.3 Memory Controllers

The memory interface is a crucial component of any high-performance processor and all the more so for a multi-core processor as it is a shared resource between all the cores on the chip. In recent high-end chips from both AMD and Intel, the memory controller was moved onto the chip and is separated from the I/O-interfaces in order to increase the memory bandwidth and to enable parallel access to both I/O devices and memory.

Of particular interest is the DRAM controller where the focus during the last few years has been to provide for increased throughput rather than low latency. DRAM request schedulers do not maintain a FIFO ordering of requests from processors to the DRAM. Instead, they try to combine accesses to the same DRAM page if possible, in order to best utilize open pages and avoid unnecessary switching of DRAM pages [4]. For a multi-core system, one can get both synergies as well as interferences in these aggressive DRAM schedulers [5].

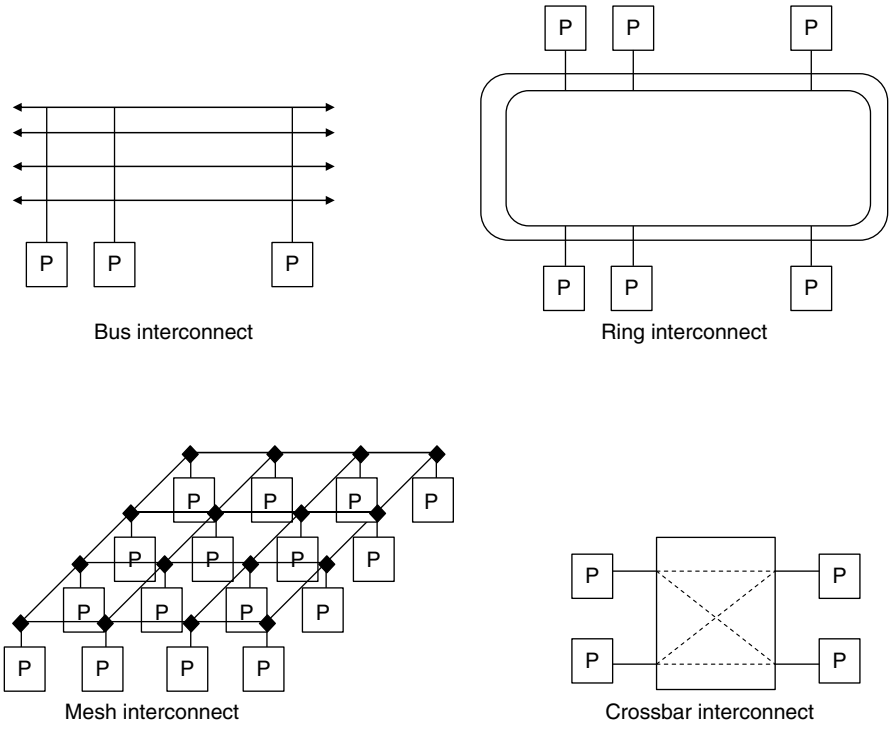


Fig. 2.1 Types of on-chip interconnect

One would expect synergies in the case of parallel programs that divide the work in a data-parallel fashion. In such programs, threads executing on different cores tend to work on the same instructions and with data located close to other threads' working-set and therefore the likelihood that they will access the same DRAM-pages is increased. On the other hand, when multi-core processors become many-core, we will see space-sharing in conjunction with the time-sharing that takes place in today's operating systems. This also happens if multi-core processors are used for throughput computing with multiple sequential applications rather than using them for parallelism. In these cases, the different applications will fight for the shared resources and may severely interfere with each other.

The memory controller is also the place where sophisticated pre-fetch mechanisms are usually implemented.

On the other hand, better utilization of increased memory bandwidth also leads to increased thermal development in the shared DRAM. Therefore, recent advances have brought forward both specialized DRAM DIMM modules for improved power consumption performance and memory controllers with DRAM throttling to reduce excessive temperature [6].

2.2.1.4 Shared Memory Support

With shared memory we mean that the processing cores can communicate with each other storing data in shared memory locations and subsequently reading them. The actual communication takes place over the interconnection network as discussed earlier.

A shared address space facilitates migration from a sequential programming model to a parallel one, in that data structures and control structures in many cases can be kept as those were in the sequential program. Programming models such as OpenMP [7] and Cilk [8] also allow for an incremental parallelization of the software, rather than the disruptive approach needed for a message-passing approach. That said, shared memory does not come for free, particularly when the core count goes up.

In the high-performance computing area, shared memory has generally not been used since its current implementations do not scale to the thousands or even hundreds of thousands of nodes used in the top-performing compute clusters today. These clusters, however, are built out of shared memory nodes and although programmers may or may not use shared programming models within a node, the hardware of these nodes typically implement a shared address space. The best performing programs in these machines typically use a hybrid programming model using message-passing between nodes and a shared memory within a node.

For these reasons, all general-purpose multi-core processors today support a shared address space between cores and maintain a *cache-coherent memory system*. By definition, a cache system is said to be coherent if and only if all processors, at any point in time, have a consistent view of what is the last globally written value to each location.

Over the time, various cache organization architectures have been proposed, relying on private, shared or mixed, flat or hierarchical cache structures. Figure 2.2 gives a schematic overview of various cache architectures.

The cache coherency mechanism allows processors fast access to commonly used data in their own private caches while still maintaining consistency when some other processor updates shared variables. The most common implementation uses invalidation mechanisms where local copies are invalidated if a core updates a shared variable. Figure 2.3 illustrates this basic principle of cache coherence through a simple example of access to shared memory.

In the example in Fig. 2.3 we have three cores, each with a private level 1 (L1) cache. For simplicity we only show the data cache, although the cores also typically have separate private instruction caches as well. There is also a shared level 2 cache unified for data and instructions. For now we assume that the cores and the L2 cache are interconnected through a shared bus.

Consider the case where both core 0 and core 1 have read the value of address A and thus both have a copy of A in their L1 caches. The need for cache coherence arises when either core 0 and 1 needs to update the value of A and the other core subsequently wants to read it. The example at the bottom of Fig. 2.3 illustrates

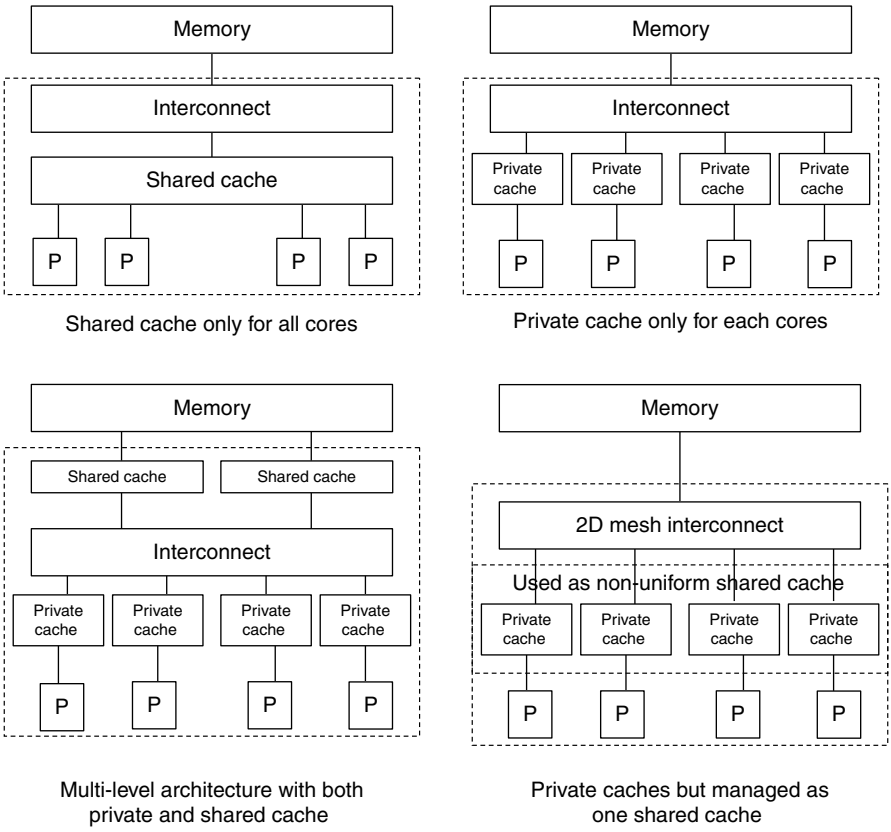
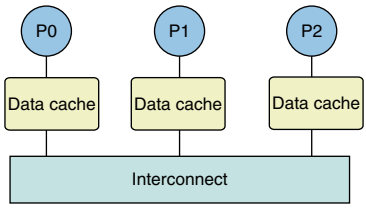


Fig. 2.2 Examples of cache architectures

a sequence of accesses to block A that will lead to some coherence actions to be performed.

The coherence actions taken on a cache block is governed by a set of state-machines. The different coherence protocols employed are often named after these states. An example of a commonly used cache coherence protocol is the MESI protocol, deployed in Intel processors. MESI stands for the four states a cache-block can be in according to below:

- **M**odified—A block in this state is the only valid copy of the block. The memory does not hold valid information and no other cache may have a valid copy. The core that owns this block can write to it without notifying any other core.
- **E**xclusive—The first core to read in a block from memory will load it into the Exclusive state. This means that if the same core later modifies the block, it can silently be upgraded to the modified state without notifying anyone else. This is beneficial for the vast majority of data which is not shared between threads or processes. If a block is in the exclusive state we know that



Initial state of block A is un-cached.

Time	P0	P1	P2	Comments
t1	Load A	Load A		A copy in both cache 0 and 1. The block is shared.
t2	Store A			The copy in cache 1 needs to be invalidated. The block is exclusive in cache 0.
t3	Load A			This load will hit in the cache and no coherence action is needed.
t4		Load A		The newly updated value in cache 0 is transferred to cache 1. The block is shared. Memory may or may not be updated depending on cache coherence protocol used.
t5			Load A	Block is now also cached in cache 2.
t6		Store A		The ownership is transferred to P1. The block is now exclusive in cache 1. The copies in both cache 0 and cache 2 are invalidated.

Fig. 2.3 Example of cache coherence in practice

  - the memory has an up-to-date copy
  - there are no other cached copies in the system.

- **S**hared—As soon as a second core is reading the same block, it will be loaded to the cache of that core and will be marked Shared in all caches
- **I**nvalid—As soon as one of the copies is modified by one of the cores, all other copies will be marked invalid and will need to be refreshed at the next access

MESI is just one of many cache coherence mechanisms proposed and implemented over time. Our goal is just to introduce the basic concepts; for a thorough overview of the area, we recommend one of the well-established books in computer architecture, such as Ref. [9] and [10].

The concept of cache coherence makes it easy to write efficient parallel programs using threaded or task-centric programming models. However, it also adds to the complexity of the memory hierarchy and power consumption. Clearly, for a parallel program to be scalable, the use of shared resources—be it memory locations, interconnects or memory interfaces—leads to performance bottlenecks. We will elaborate on this in the section dedicated to scalability issues later on.

2.2.1.5 Memory Consistency

The problem of consistent view of the memory is one of the fundamental problems that need to be tackled in any multi-core design. The existence of multiple copies of the same physical memory location—at various levels of caches but also within processor cores—requires a consistent and easy to understand model of how concurrent loads and stores are coordinated in order to maintain a consistent view of the content of the memory.

One of the important concepts related to memory consistency is *store atomicity*. In the presence of multiple copies of the same memory location, a store to that memory location needs to be propagated to all cores in *zero time*—something that's impossible to achieve in practice. However, the *appearance of instantly propagated store* can be achieved if a global order of store operations to the same memory location is enforced and a load (use of the value) is globally performed before the value it returns is used, meaning that the store providing the new value is performed with respect to *all cores* in the system.

The strictest memory consistency model available today is called *sequential consistency*. Formally, it is defined as follows:

> A multiprocessor is sequentially consistent if the result of any execution is the same as if the memory operations of all threads were executed in some sequential order and the operations of each individual thread appear in thread order.

Intuitively, it means that, while the order of memory accesses with respect to one core is maintained, accesses from different cores may be interleaved in any order. If one would perform a poll among programmers, this would be the most popular choice of memory model—but it requires significant implementation effort in hardware.

Several other models have been proposed over time, with weaker consistency requirements. Here are a few examples:

- allow a load to bypass a store within a core, if it addresses a different location in the memory
- allow stores issued from a core to be used by subsequent core-local loads even before it's globally visible
- rely on the use of atomic sections when accessing shared memory areas, thus enforcing mutual exclusion; all other loads and stores are performed without any consistency enforcement, as these are considered core local
- processors relying on out of order execution sometimes deploy speculative violations of memory orders: the core will *assume* that the values read or written will not conflict with updates from other cores; if this assumption turns out to be wrong, the operation is rolled back and re-executed; however significant performance gains can be achieved in case no conflicts are detected

While taken for granted by programmers, memory consistency is one of the most critical and complex issues to be considered when designing multi-core systems. Understanding the fundamentals of how memory access works is therefore essen-

tial when dealing with tricky concurrency bugs or implementing support for basic synchronization primitives.

2.2.1.6 Atomic Operations

As we will see further on in this book, *partitioning* into parallel tasks and *synchronization* between tasks are the fundamental activities that are required when designing software for many-core systems. *Synchronization* is very hard to realize in software alone, thus support in hardware is required; however, hardware only synchronization is usually hard to scale and has limited flexibility. Therefore the most common solutions rely on software with basic support from the hardware. We will look at the software solution in Chap. 5; here we will focus on the support available in hardware.

The mechanism that modern processors provide in order to support synchronization mechanisms are so called *read-modify-write (RMW)* or *conditional stores*. The basic principle behind these instructions is to provide the *smallest critical section* that guarantees conflict free access to a specific memory location that will contain the value used for synchronization. The most commonly used RMW instructions are listed below:

- *Test and set (T&S):* it reads a memory location, sets it to 1 and returns the value read in a register atomically (thus no other core can perform any store action on this memory location while the T&S instruction is being executed). Using this instruction, the mechanism to acquire/release a lock would look like this:

```
Locking:
   register = test_and_set (lock);
   while (register != 0)
         register = test_and_set (lock);
Unlocking:
   lock = 0;
```

- *Compare and swap (CAS):* atomically compares the value in the memory to a supplied value and, if these are equal, the content of the memory is swapped with the value stored in a register. Lock implementation using CAS looks like this (quite similar to T&S based implementation):

```
register = 1;
compare_and_swap (lock, 0, register); // if lock is 0, swap its value to 1
while (register != 0)
    compare_and_swap (lock, 0, register); // if lock is 0, swap its value to 1
```

- *Load-linked and store-conditional:* this is a de-coupled version of T&S, which is more pipeline-friendly. While there are two instructions in this case, the two are linked: after a load-linked, a store-conditional will only succeed if no other operations were executed on the accessed memory location since the execution

of the load—otherwise will set the register used in the store-conditional to 0. Here's how locking can be implemented with these mechanisms:

```
register = 0
while (register == 0) {
  dummy = load_linked (lock);
  if (dummy == 0) {
    register = 1;
    register = store_conditional(lock, register);
  }}
```

The presence of one of these ISA level constructs is essential for implementing higher level synchronization mechanisms. However, just one of these basic mechanisms is sufficient for most flavors of software synchronization primitives as well as for realization of lock-free data structures.

2.2.1.7 Hardware Multi-threading

As the capabilities and speed of cores kept improving at a more rapid pace than that of memory, the net result was that cores were idling for a significant share of the time while waiting on high latency memory operations to complete. This observation led to the implementation of *hardware multi-threading,* a mechanism through which a core could support multiple thread contexts in hardware (including program counter and register sets, but sharing e.g. the cache) and fast switching between hardware threads whenever some of the threads stalled due to high latency operations.

Various implementations of this concept have been provided over time, but most present systems use a fine-grained multithreading approach in the context of out-of-order cores called *simultaneous multi-threading.* In each cycle, instructions from different hardware threads are dispatched, or—in case of super-scalar processors where multiple instructions can be dispatched per core cycle—even instructions from different threads may be executed in parallel. As instructions from different threads use different registers, the latency of detecting where out-of-order execution can be performed is significantly reduced, leading to a higher density of instructions effectively executed per cycle. The goal of this technique is ultimately this: assuming that the software is multi-threaded, exploit it to mask the slowness of fetching data from memory and improve core utilization.

This technique has been implemented by most major vendors, including Intel (through the Hyper Threading Technologies (HTT) concept), IBM (which also supports thread priorities) and Oracle Sun (where as much as eight hardware threads are supported on each core).

Few years ago, there were arguments brought forward [11] that suggested that hardware threads actually consume more power than similar designs based on multiple cores, while also increasing the frequency of cache trashing. By 2010

however most chip vendors either already supported it or announced plans to do so.

The usefulness of this technology also has its limit however. Recently, the gap between the speed of memory access and speed of cores started to narrow due to the decline in processor core frequencies; therefore latency-hiding techniques will yield smaller benefits. Considering the performance gain per watt, it's likely that these techniques will be replaced by other mechanisms, such as increased cache sizes.

2.2.1.8 Multi-processor Interconnect

Multi-processor, cache coherent interconnect is an important building block for today's multi-processor, NUMA systems: it allows linking multiple processors together in order to provide a single logical processing unit. There are two major technologies found in today's systems:

- *HyperTransport,* a standardized low-latency packet-oriented point-to-point link supported by many major vendors (except Intel); its latest specification allows for speeds up to 51.2 Gbps/s, running at 3.2 GHz
- *Intel's QuickPath Interconnect (QPI),* used in most Intel chips

These types of interconnects are also used for connecting with I/O devices; however, in the context of this book, their importance lies in the possibility of creating more complex, higher core count logical processors from simpler building elements, with just a few cores.

2.2.1.9 Summary

In this section we introduced the well established concepts used in the design of the state of the art multi-core systems. There's a red line throughout these concepts that can be summarized in a few principles and assumptions:

- The only way forward is to start adding—albeit slowly—more cores to future chip designs
- These cores shall still be quite complex and high performance to support single threaded applications
- Shared memory on hardware level will still be required, even as we scale up the number of cores per chip
- The main issues to address are still memory access latencies, hence we need continued support for cache hierarchies and mechanisms that can improve efficiency of core usage

Only recently the community started questioning some of these principles and assumptions, which we will discuss later on in this chapter. It's therefore useful to look at some of the other ideas that are slowly making their way into the design of multi-core processors.

2.2.2 Emerging Concepts

As multi-core chips became ubiquitous, several new issues emerged that required new solutions, while some existing, well-known problems have found novel solutions. In this chapter we will look at some of these concepts.

The problem of *memory wall,* initially defined as the gap between the speed of processors and the speed of memory access, has slowly morphed into a different problem: the latency gap became smaller, but with the increase in the number of cores, the need for memory bandwidth has increased. Interconnects became the other issue that needed attention: existing solutions became either too power hungry as the transistor count went up (the case of bus or ring solutions) or led to higher latency (mesh networks), as the number of cores continues to increase. Finally, the larger number of cores starts to question the emphasis on core usage efficiency and ideas relying on using some of the cores (or at least hardware threads) as *helpers* have popped up recently.

On software side, as the number of cores increased, the cost of pessimistic, lock-based synchronization of access to shared memory got higher, prompting the search for new mechanisms, leading to the emergence of the concept of *transactional memory.*

2.2.2.1 Scouting and Helper Threads

The idea of scouting hardware threads was promoted by Sun as part of the design of their new processor called Rock (canceled since). The idea is to let the execution of a hardware thread continue even if normally it would stall due to e.g. a memory access: while waiting for the data to be fetched, the processor switches to a mode called *scouting* and would execute those instructions—out of order—that are not dependent on the result of the instruction that caused the stalling. Once the execution of the high latency instruction completes, the execution is re-synchronized and continued from the next instruction that was left out by the scouting process (due to data dependency or latency of execution). This technique could be augmented with speculative execution: the processor could speculate on the likely outcome of the stalled instruction and use this assumption for the scout thread's run-ahead execution.

The scout thread idea clearly targeted the traditional memory wall problem, trying to mask the latency of memory accesses. We believe that in this form and under this assumption (latency is the primary concern) it is a technique of diminishing returns; however the idea itself—using helper threads to speed up execution—has its merits and alternative use cases have been proposed.

Paper [12] proposes the usage of helper threads to improve the efficiency of cache usage: a separate core (or hardware thread) would monitor the memory traffic to and from a specific core, recording memory access patterns; using this information, whenever the same pattern is observed again, the helper core would initiate fetching of data from memory to cache, thus pre-loading the cache ahead of time. If the data is already in the cache, the helper core could make sure that it stays there and no unnecessary write-backs would occur. This technique tends to reduce both

latency, but also optimize memory bandwidth usage: if the prediction is correct, useful memory traffic is prioritized and unnecessary one can be avoided.

A similar idea is proposed in Ref. [13] to cache invalidation (trashing). In this case, the execution of the OS would be offloaded to a special core, so that the content of the cache relevant for the application would not be destroyed through the loading of the data needed by the OS. This idea is similar to the factored OS principle that we will present in Chap. 7.

Thread Level Speculation

Thread level speculation support in hardware is quite similar to the scouting thread concept: the processor is capable of performing run-ahead execution on certain branches, using private copies of data and registers, at the end either validating or discarding the result. This idea may get more traction as the number of cores will continue to increase: some of the cores can be used to perform speculative execution on behalf of the main branch of execution. Such an approach suits well heterogeneous single-ISA multi-core architectures: the main thread of execution is placed on the high-capability core, while the simpler cores will be responsible for low power speculative pre-execution.

These ideas have not yet made their way into actual products, but as we will see later on in this book, such an approach could provide a way forward for scaling hard to parallelize applications on many-core processors.

2.2.2.2 Memory-centric Architecture

It's becoming clearer that memory bandwidth rather than memory latency is the next bottleneck that needs to be addressed in future chips. Consequently we have seen a dramatic increase in the size of on-chip caches, a trend that is likely to continue as the amount of logical gates on a chip will keep increasing for some time to come, still following Moore's law. There are however several other directions based on novel technologies that are currently pursued in the quest of improving memory bandwidth:

- Embedded DRAM, already in production in commercial chips such as IBM's Power7 processor
- Use of 3D stacking of memory with short, high bandwidth links called vias between processor cores and memory
- Use of memristors, perhaps the most radical departure from today's designs

Embedded DRAM

DRAM has the benefit of having much higher density and lower power consumption than the traditional SRAM technology used for on-chip memory, thus integrating it on the same silicon with the processing logic holds the promise of higher

on-chip memory density and higher bandwidth access than through external interfaces. Novel developments in CMOS technology allow today the usage of the same process to manufacture both memory and processing units.

Perhaps the most well known usage of embedded DRAM is in IBM's Power7 processor, packing a 32 Mb L3 cache built using eDRAM; however the technology has now been deployed in various gaming and embedded processors as well.

3D Memory Stacking

Packaging memory on top of the processor cores allows the creation of very high speed, dense interconnects (*vias*) between the two layers, achieving two goals at the same time: dramatically increasing the amount available memory (to the order of gigabytes) with access speeds comparable to cache access today.

Beside practical implementation of layering itself, this technology has a number of challenges related to cooling the processor cores hidden under the memory layer. The concept is still in research phase with a number of prototyping activities ongoing at universities and industrial research laboratories. If realized, it will allow a dramatic improvement in memory performance.

Memristor-based Technologies

The concept and theory of *memristor* (short for memory resistor) has been formulated almost four decades ago [14], but only as recently as 2008 the first practical realization was announced by HP Labs. In short, it's a circuit element with "memory": the resistance is a function of past current, in other words, a memristor can remember a voltage applied to it even after the current is cut off. This property makes it a prime candidate for building more compact and faster storage devices, while computer designs based on memristors replacing transistors have also been proposed. If it will materialize, it will represent a unified design base with significant possibilities: 3D designs based on memristors with up to 1,000 layers have been designed.

Memristor based commercial processors and memory systems are still some time—perhaps a decade—off, but currently this is one of the most promising basic technologies in the quest for improving memory performance.

2.2.2.3 Future Interconnects

As the number of cores on a chip will continue to increase, the on-chip interconnect is quickly becoming the bottleneck, both in terms of throughput and power consumption. This is especially true for bus and ring based interconnects: the power needed to drive high-speed networks across long wires is quickly becoming the largest contributor to overall power consumption, outstripping all other components (such as cores and peripherals). Mesh/switch based interconnects, due to shorter wiring were able to keep power at manageable levels; however, as the size of the

mesh will continue to increase, the latency and delay of transporting data across the chip will become the limiting issue.

There are two significant new ideas that aim at tackling the power, latency and delay related issues with the on-chip interconnect: 3D stacking coupled with mesh interconnect and optical interconnects. 3D stacking of cores, not just memory would help limit the number of hops that would need to be used to interconnect the cores. Consider for example the case of a 1,024 core machine: in a 2D mesh of 32×32, the cost of reaching across the chip is 62 hops (31 in both directions); if it's placed in a 3D configuration with 4 layer of 256 cores, this value drops to only 33 hops (3 to cross the layers and maximum of 30 in the lower layer), a factor of almost 3× improvement. Obviously, 3D stacking of cores has to solve the same issues that have so far hampered efforts of stacking memory on top of processor cores: cooling and heat removal from the lower layers.

Optical On-chip Interconnect

Optical on-chip networks are in the focus of recent research efforts at both universities and chip manufacturing companies such as IBM or Intel. Besides offering higher speeds, having lower power consumption and occupying less chip real-estate, optical interconnects have the possibility to span multiple physical chips, thus creating one logical chip from smaller physical chips.

Optical interconnect solutions are likely to use *Wavelength Division Multiplexing (WDM)* as underlying transmission technology, which allows a single optical waveguide to simultaneously carry multiple independent signals on different wavelengths. Optical signals use significantly less power because of the relatively low loss even across long distances.

One interesting approach was put forward by researchers at MIT [15], illustrated in Fig. 2.4. Their approach is to use a broadcast optical interconnect (ONet in the figure) between islands of several cores (16 each, in their largest configuration) and rely on island-local electric broadcast of information received over the optical network. In addition, an island-local electric mesh network would be preserved for local communication (*ANet* in the figure). This hierarchical architecture helps simplifying the architecture of each network: the optical network would have only 64 endpoints in the largest (1,024 cores) architecture while the electric networks would be a 16-way star and 4×4 mesh network, respectively.

On-chip optical interconnects have yet to make their ways into commercial chips. However, in our assessment, this technology represents the best bet for improving performance of on-chip interconnects.

2.2.2.4 Transactional Memory

Transactional memory emerged as an alternative to traditional locking based synchronization. We will discuss it in more details in Chap. 5; here we will focus on the hardware aspects of its realization.

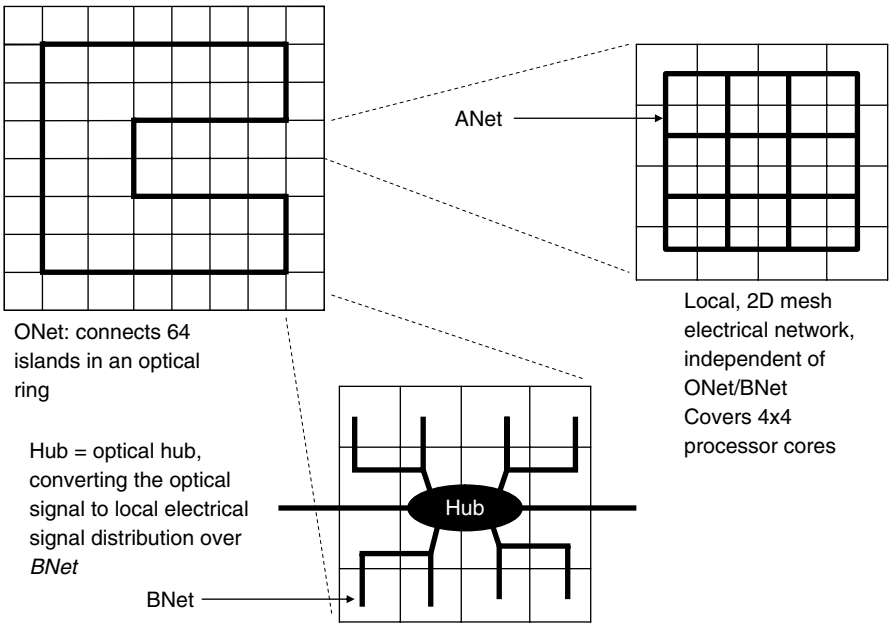


Fig. 2.4 Architecture of a combined optical/electrical on-chip interconnect

In short, transactional memory is a mechanism to enable optimistic concurrent updating of shared memory locations: a thread is allowed to progress with its own updates until it completes the sequence of actions (called *transactions*: the changes performed during a transaction are not visible to the rest of the system); at that point, it is checked if any other concurrent accesses took place on the same memory locations: if not, all the changes are *atomically committed* and made visible to other cores/threads; if the memory areas modified by the transaction have been modified elsewhere, the complete transaction is *rolled back* and re-attempted. All these mechanisms—private updates, check for concurrent access, commit/rollback/retry—are transparent to the application and handled by the run-time system. Transactional memory is most useful in situations where the probability of contention is very low.

There are three ways to implement the concept of transactional memory: in hardware only (called Hardware Transactional Memory—HTM), in software only (Software Transactional Memory—STM) or a hybrid solution, combining both hardware and software solutions.

Hardware transactional memory requires significant changes to the memory management and thread management system of the processor. Each core will have a new state—the *transactional state*—in which handling of the active thread and the memory will be different. The ISA of the processor shall support new commands for indicating the start and end of a transaction.

How will hardware transactions work?

At the beginning of a transaction, the processor has to checkpoint the state of the executing thread, i.e., the content of the registers and the memory. Registers are

easy to manage by supporting two register files: the "regular" one and the "transactional" one; during transactions, a copy of the regular one is created in the transactional one which is then either committed (copied back to the regular one) or discarded (in case of rollback) at the end of transactions.

There are multiple solutions proposed for handling memory during transactions. Here we will describe a simple method, based on a mechanism called *transactional cache state.* Basically, there shall be two copies of the blocks being edited: the version before the transaction (the stable copy) and the one being modified by the transaction (the transactional copy). The simplest way to manage the two copies is to hold the transactional copy in the L1 cache and the stable copy in L2 cache. This will require a new state of the block (beside the MESI ones): *Transactional,* indicating that the block is being modified as part of an ongoing transaction and shall not be shared with other cores. With the method described here, this new state is needed for blocks cached in L1 cache.

There are two methods to detect conflicting updates to the same shared memory: the eager method detects the conflict and aborts one of the transactions immediately; the lazy method does so only at the end of transaction. Using the modified MESI coherence protocol, the eager implementation is the more handier one: when creating the L1 copy of the block, the core will be notified if it's already under editing on an other core; in this case it can immediately abort the transaction (and retry later on).

One of the major issues with this method based on L1 caching of transactional blocks is the limited size of the L1 cache: for long running transactions or transactions that modify a lot of data, the size of the L1 cache may become insufficient to hold all transactional blocks. There's no easy way around this limitation—hence keeping the size of transactions (both time- and space-wise) short is a pre-requisite for this method to work.

Despite extensive research—the first paper on transactional memory was published already back in 1993 [16]—transactional memory support has yet to become part of modern processors. Many of the major players in the industry chose to rely on STM implementations first—the only processor that was planned to support hardware transactional memory was the now canceled Rock processor from Sun.

2.3 Scalability Issues for Many-core Processors

So far in this chapter we looked at the current well established technologies used in building multi-core processors as well as at the emerging technologies that are targeting some of the shortcomings of current technologies.

On manufacturing technology level, major chip manufacturers estimate that we will be able to continue with current CMOS technologies down to approximately the 6 nm manufacturing process (as of 2010, the most advanced technology node is the 32 nm node). However at that level at least two major challenges will emerge:

- As the size of transistors will be measured in just a few tens of atoms at most, quantum effects will have to be taken into account and consequently we will see

an increased *unreliability of the hardware*, with components failing more often and—more importantly—intermittently
- There will be so many transistors on the chip that it will be, power wise, impossible to switch all of these at the same time; this phenomenon—called the *dark silicon* problem—will have a significant impact on how we will build future processors

The unreliability of future hardware will likely lead to the implementation of redundant execution mechanisms. Multiple cores will perform the same computation, in order to increase the probability that at least one will succeed; in some cases a voting scheme on the result (verifying if all the computations yielded the same result) may be used to guarantee correctness of the calculation. Such mechanisms will likely be invisible to the software, but will impact the complexity of the design of logical processor cores.

The dark silicon problem is trickier to address. By lowering the frequency at which chips operate, we can push the limit further [17], but eventually it will become an issue, no matter how low we go with the frequency. Alternative solutions include the adoption of memristors as building blocks (instead of transistors) and chip designs where, at any given time, just a subset of components would be active, depending on the type of application. Optical interconnects within and between chips could also allow building smaller chips, interconnected to build larger, cache coherent logical chips.

Another scalability bottleneck relates to the design of cache coherency protocols. Synchronizing access to the same memory area from large number of cores will increase the complexity of coherency design and will lead to increasing delay and latency in accessing frequently modified memory areas. In our view, it will be an uphill battle to maintain a coherent view across hundreds, let alone a thousand cores; even if we will be able to do this, it will be hard to justify the cost associated with it.

Memory bandwidth will be another scalability bottleneck. The leveling out of the core frequency will lead to reduced latency, but the increase in the number of cores will multiply the amount of data required by the cores and thus the aggregate memory bandwidth that future chips will require. If we will indeed see the continuation of Moore's law, translated into an ever-increasing number of cores—perhaps following the same trend of doubling core-count every two years—a similar trend would need to be followed by memory bandwidth, something the industry failed to deliver in the past.

Based on these observed or predicted developments and bottlenecks, we believe the following trends will dominate the design of future processors:

- Shift towards *simple, low frequency, low complexity* cores, coupled with an increase of the core count to the level of several hundreds within five to ten years; heterogeneity—not in ISA, but rather in core capabilities—will play a role simply because it's an easy optimization gain
- Focus on *novel memory technologies* that can deliver higher bandwidth access; technologies such as 3D stacking, optical interconnects and perhaps memristors

will see an accelerated uptake by mainstream processor designs; especially optical interconnects have the potential of easing some of the pressure on how chips are structured and interconnected
- The size of *on-chip memory* will also see a dramatic increase and we will see innovations emerging that will reduce the footprint, power consumption and complexity of designing such solutions, similar to the development of the embedded DRAM technology; once again 3D stacking and memristors may be some of the technologies to watch
- *HW accelerators* will be abundant: these have low footprint and low power consumption, thus we will see realizations in HW of an increasing array of algorithms
- *Aggressive power optimization mechanisms*—near threshold operation, power gating, frequency and voltage scaling—will be pervasive not only in traditionally low power domains, but also in most areas where processors are used

Some of these predictions may fade away, but, in the absence of a revolutionary new method of designing processors, increasing core count, heterogeneity and reliance on aggressive power optimization methods will likely dominate the chip industry for the coming five to ten years.

2.4 Examples of Multi-core Processors

After following the path of predictable design—faster, more complex, single-core RISC (Reduced Instruction Set Computer) or CISC (Complex Instruction Set Computer) machines—for about 30 years, the chip industry has seen an explosion of wildly differing designs over the past seven years. If anything, this trend will continue well into the second decade of our century—thus any attempt to survey the landscape of representative designs is likely to lead to outdated results very quickly.

Therefore in this chapter we will try to showcase the major trends followed by processor vendors rather than individual incarnations of certain processor families; whenever we delve into the details of a specific processor, the goal is to highlight and analyze a certain design choice, rather than upholding that specific version of the chip.

There are essentially three different philosophies followed by multi-core chip designs:

- fewer, but more complex, high performance cores with large, shared on-chip caches and cache coherence mechanisms; this design is represented by most server chip vendors (IBM, Intel, AMD and SUN to some extent) but also companies focusing on the mobile and low power design space
- large number of simple, low frequency, sometimes specialized cores with distributed on-chip memories (with or without cache coherence mechanisms) and scalable interconnects; these chips usually target data-parallel applications such

as graphics processing or networking, with the notable exception of Tilera which targets a broad range of applications, including data-centers
- integration of multiple cores of different capabilities and potentially supporting different instruction set architectures

The main argument for the first type of design is support for single-threaded applications. Multi-core is regarded a "necessary evil"—the next best thing once performance gains through higher frequencies could not be sustained. On the other hand, the advocates of the second approach rely on the argument that overall performance per watt will be higher using simpler but more cores. The third type of design usually targets specific domains where some measure of performance—overall, per watt or per real estate—dominates all other factors.

We tend to agree with the second line of thinking—the first type of approach is limited to just incremental improvements; without novel programming models that would allow running even single threaded applications on multiple cores, this approach will eventually not be sufficient.

2.4.1 Processors Based on Low Number of Cores

In this chapter we will cover the most representative chip families where the single-core performance still is the primary design concern: Intel's server processors (based on the Nehalem microarchitecture), IBM's Power series of processors, Oracle/Sun's SPARC ISA-based processors as well as designs based on ARM's Cortex-A15 core design.

2.4.1.1 Intel's Server Processors

Intel's current (as of 2010) line-up of 64 bit server processors relies on the *Nehalem* micro-architecture, first introduced in 2008 and revised in early 2010 (under the code-name *Westmere*). The successor of this microarchitecture will be released in 2011, under the code name *Sandy Bridge*; the major change will be the integration of dedicated graphics cores. Today chips based on this micro-architecture are manufactured using 45 and 32 nm processes (which will be used for the *Sandy Bridge* line as well, at least initially).

The main features of this line of processors include:

- Native support for up to 8 cores/die (typical configurations have 4 or 6 cores)
- Integrated memory controller
- Support for shared L3 (missing from previous generations) with sizes up to 12 Mb, in addition to per core L1 and L2 caches
- Support for virtualization (dedicated protection ring)
- Support for Intel's Turbo Boost, Hyper-Threading, Trusted Execution, and SpeedStep technologies

- Core speeds of up to 3.33 GHz, with a maximum turbo frequency of about 3.6 GHz
- Maximum power consumption (TDP—thermal design power) of 130 W, including on-chip, but off-processor GPU cores, with 6 processor cores running at 3.33 GHz.

Clearly this family of processors is targeting server workloads with special emphasis on single-threaded applications and power management. From technology point of view it's important to understand the background of different technologies supported by Intel's server chips, the focus of the following sub-sections.

Turbo Boost

Turbo boost is Intel's implementation of dynamic frequency scaling. This technology allows cores to run faster than the base operating frequency, if the overall power and temperature specifications allow it. It must be explicitly activated, but the actual operating level is determined by three factors:

- Number of active cores
- Estimated power and energy consumption
- Processor temperature

The operating frequency of the core is raised stepwise, with 133 MHz at a time, at regular intervals. The number of steps depends on the conditions (especially the number of active cores) and the base frequency, the maximum range is typically 10–15% compared with the core base frequency.

Hyper Threading

The hyper-threading technology is essentially the implementation of hardware threading, provided by Intel all the way from the low power Atom processor till the top tier server chips. The support is limited however to only two threads per core. There's no priority assignment to threads and the hardware will make sure that no thread gets starved.

Trusted Execution

Intel's trusted execution technology provides hardware support for enhanced security of program execution. Specifically, it allows for

- Creation of multiple, isolated execution environments (partitions) that are guaranteed to be tamper proof by any other application running in an other partition
- Secure key generation and storage
- Remote attestation of the security level of the machine

These features are usually realized as a combination of processor features and support in chipset and firmware.

2.4.1.2 IBM's Power Processors

IBM's line of POWER processors—now at its 7th generation—is based on the Power Architecture instruction set architecture, driven primarily by IBM and Freescale, within the Power.org industrial consortium. The ISA has a RISC architecture and it's open for licensing. Despite the pervasive nature of the X86 ISA (on which Intel and AMD processors are based) in the server space, roughly half of the Top 50 supercomputers are based on processors using the Power Architecture (all built by IBM). Additionally, the Power architecture is available in many specialized chips, including almost all game console processors.

The main characteristics of the Power 7 family of processors are:

- 4, 6 or 8 cores per chip, with execution frequencies exceeding 4 GHz; a more conventional limit is around 3–3.5 GHz—which still puts power consumption over the 200 W bar
- 4-way SMT, resulting in 32-way SMT per processor, with aggressive out of order execution
- 32 Mb on chip, embedded DRAM based shared L3 cache, on top of the 64 kb/core L1 and 256 kb/core L2 cache, tightly coupled with the cores
- Scalability up to 32 sockets, with "near linear" performance scaling claimed by IBM
- Advanced power optimization designs
- Distributive resource management which allows re-allocation of resources (cache and external memory bandwidth) between cores, depending on the application that is being executed

Power7 processors stand out—in our opinion—with two design choices: the usage of eDRAM (which we have discussed elsewhere in this chapter) as basis for L3 cache and the advanced power management features.

The cores in the Power7 processors support two idle modes:

- *nap mode*, optimized for wake-up time: clocks are turned off towards the execution units, frequency is reduced, but caches and virtual memory management tables remain coherent, thus the core can be brought back to full speed quicker
- *sleep mode,* optimized for power reduction: clocks are fully turned off, caches are purged and voltage reduced to a level where leakage current is substantially lowered; however, at wake-up still no re-initialization of the core is required

The processors also support active energy management, commercially called Energy Scale, consisting of the following technologies:

- DVFS (dynamic voltage and frequency scaling) within the range of −50% to +10% of the nominal core frequency

- Turbo mode, similar to Intel's solution
- Power budgeting for different parts of the system: performance will be optimized within pre-configured power limits

The Power 7 family of processors is a fine example of the balancing act server chip vendors have to perform in order to deliver chips that can support both single-threaded as well as multi-threaded workloads. Features such as high clock rate (the highest of any chip delivered during 2010), turbo mode, distributive resource management are geared towards supporting single-threaded performance; DVFS (Dynamic Voltage and Frequency Scaling), the relatively large number of cores and the distributed cache architecture have as a primary goal support for parallel scalability.

2.4.1.3 SPARC Processors

Beside the two X86-based vendors and IBM, Sun Microsystems (acquired by Oracle) was one of the long-standing leaders in the area of server chips and high end servers, with designs based on the SPARC instruction set architecture, initially developed at Sun. Recently however Sun also diversified itself into supporting X86 based architectures.

SPARC (Scalable Processor Architecture) is an open, RISC type of instruction set architecture, licensable in a similar manner as the Power specification; several of the processors designs based on SPARC have actually been released under open source license (such as OpenSPARC T1 and OpenSPARC T2). One of the SPARC architecture based processors also serves as the baseline for the widely used SPEC (Standard Performance Evaluation Corporation) CPU benchmarks: all benchmark results represent a relative speed compared to that of the basic SPARC processor; for more details on the SPEC benchmarks [18]. As of 2010, there are essentially only two large vendors providing chips based on the SPARC specification: Fujitsu and Oracle/Sun.

An interesting feature of the SPARC ISA is the support for *register windows*. The processor may have up to 160 general purpose registers, but at any time only 24 of these are visible to the software; whenever a sub-routine call is performed, the register window is shifted by 8, so that the new procedure gets 8 local registers and shares 8 registers with its caller as well as another 8 with any other sub-routine it may call.

The latest processor released by Sun, based on the SPARC version 9 architecture is the SPARC T3, code-named Rainbow Falls or Niagara 3 (all UltraSPARC processors used the Niagara code-name—the tag *Ultra* was dropped in this case). It's the server chip with most parallelism: it contains 16 cores running at 1.65 GHz, each with support for 8 hardware threads per core, thus the chip supports 128-way SMT. The schematic layout of the chip is shown in Fig. 2.5.

What sets the SPARC T3 apart is its on chip and off chip interconnect solution. The basic connectivity between cores and L2 caches is through a cross-bar type of

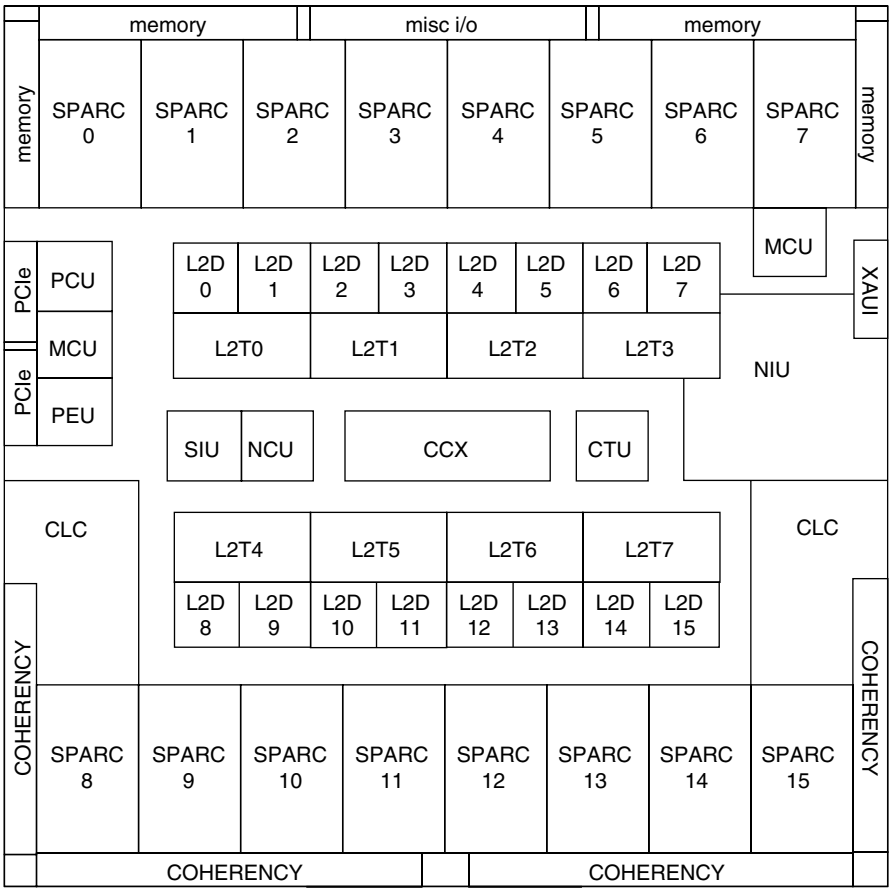


Fig. 2.5 SPARC T3 architecture. (Source: Wikipedia, under Creative Commons Attribution 3.0 license)

interconnect that connects the cores to a total of 6 Mb of L2 cache (organized into 16 banks). On the external side, there's a wide array of connectivity options: two PCIe interfaces, 2 Ethernet interfaces with a bandwidth of 10 Gbps each and 6 coherence links per core, each with a bandwidth of 9.6 Gbps. This architecture allows gluing together, without any additional hardware, four T3 chips in a cache coherent, SMP structure—resulting in a system with 64 cores and 1,024-way simultaneous multi-threading support.

The SPARC T3, while clearly a server chip, is targeted to a specific type of workload: large number of relatively simple parallel tasks that require a lot of traffic, typical for web servers and database systems (it's not surprising that offerings based on T3 are tightly integrated with Oracle's database). Its focus is a different market from Intel's and IBM's, whose chips are primarily geared towards more computationally intensive applications with less inherent parallelism.

2.4.1.4 ARM-based Multi-Core Processors

If X86 and Power architecture based systems dominate the server and desktop space, the mobile and low power computing space has its own pervasively present player: most mobile phones and a lot of embedded devices today use a chipset based on the ARM architecture, licensed by the UK based company of the same name.

ARM itself does not develop chips: its business relies on designing and licensing aggressively power optimized core designs that can then be integrated into actual chips by one of its licensees. Consequently, there's a large variety of chips based on ARM cores, all sharing the same basic ISA and core design.

Traditionally ARM cores were simple, low power designs unsuitable for desktop, let alone server usage; recently however ARM entered a new territory with its Cortex A9 and A15 designs that added capabilities previously only seen in high end desktop and server products. For this reason we include ARM into the category of chips with fewer, yet more complex cores.

ARM Cortex A15 Architecture

The ARM ISA is a 32 bit RISC architecture with fixed instruction width and mostly single cycle execution. It features some more peculiar features such as support for conditional execution of instructions (the result of the last comparison can be used as conditional for subsequent instructions) and support for folding bit-shifting instructions into arithmetic ones. It has been extended with optional ISAs covering floating point operations, SIMD instructions and native execution of Java byte-code (called *Jazelle*).

The Cortex A15 core design (announced in 2010, expected to ship in 2012) integrates all these features into an out-of-order, speculative issue, superscalar architecture with cores clocked between 1 and 2.5 GHz. The licensable solution will support up to 4 cores in a cluster, with the possibility to link two clusters into one cache coherent SMP system. For the first time for ARM, A15 has virtualization support (with a new protection ring) and support for addressing more than 4 Gb of memory (the address space is extended to 40 bits). The cache architecture will feature 64 kb L1 cache and up to 4 Mb shared L2 cache for the 4-core cluster.

It's yet unclear how chips based on the A15 core design will measure up against competing products. ARM claims a significant improvement compared to the previous generation and academic evaluations have shown Cortex A9 (the predecessor of A15) based chips to outperform—in terms of performance per watt—competing Intel server chips by a factor of about 5–7× ([19]). It all depends however on the type of the application: ARM based designs will likely perform much better with tasks that are highly parallel and do not require execution of lengthy single-threaded code.

The main differentiating factor for ARM based processors is power efficiency. While other vendors are trying to retrofit their high-performance designs with power efficient solutions, ARM cores are trying to ramp up performance on a basic

design that is eminently power efficient. So far, for embarrassingly parallel applications, the second approach seems to be gaining the upper hand.

2.4.2 Processors Based on Large Number of Cores

When looking at raw performance alone (total number of instructions executed within a unit of time), more but less powerful cores clearly outperform chips with few but powerful cores, within a set power budget. This is a tempting prospect, especially in domains with abundant parallelism such as networking, graphics, web servers etc. Accordingly, this category of chips—with many, but simpler cores—is usually represented by processors targeting a specific domain.

In this chapter we survey some of the best known representatives of this category: Tilera's Tile GX family, NVIDIA's Graphics Processing Units (GPUs), picoChip's 200-core DSP as well as Intel's recently announced Many Integrated Core (MIC) architecture.

2.4.2.1 The Tile Architecture

The Tile architecture has its origins in the RAW research processor developed at MIT and later commercialized by Tilera, a start-up founded by the original research group. Chips from the second generation are expected to scale up to 100 cores based on the MIPS ISA and running at 1.5 GHz, within a power budget of under 60 W.

The key differentiating technology of the Tile architecture is the on-chip interconnect and the cache architecture. As the name implies, the chip is structured as a 2D array of *tiles,* where each tile contains a processor core, associated L1 (64 kb) and L2 cache (256 kb per core) and an interconnect switch that can connect the tile to its 3 (on the edges) or 4 (inside the mesh) neighboring tiles. This way, the interconnect is essentially a switched network with very short wires connecting neighboring tiles linked through the tile-local switch.

The interconnect network—called *iMesh* by Tilera—actually consists of five different networks, used for various purposes:

- application process communication (UDN)
- I/O communication (IDN)
- memory communication (MDN)
- cache coherency (TDN)
- static, channelized communication (STN)

The latency of data transfer on the network is 1–2 cycle/tile, depending on whether there's a direction change or not at the tile. The overall architecture of the Tile processor concept is shown in Fig. 2.6

This network based architecture allows for some innovative solutions. The network of L2 caches is organized into a non-uniformly structured L3 cache, using a

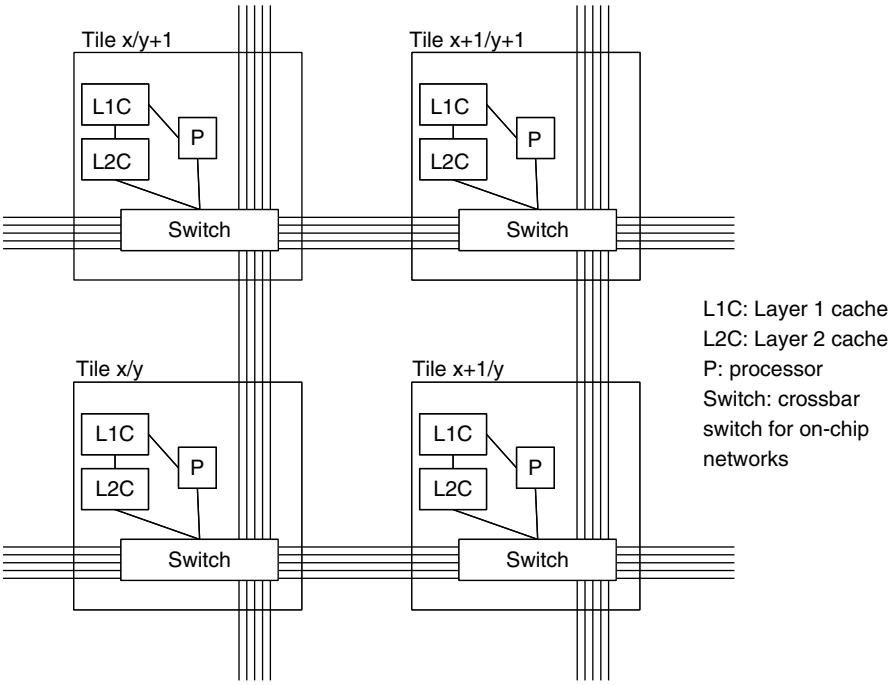


Fig. 2.6 The Tile architecture

directory-based coherence mechanism and the concept of *home tile* (the tile that holds the master copy) for cached data. While the access latency to various parts of this virtual L3 cache varies according to the distance between tiles, this mechanism provides an efficient (space and power wise) logical view to the programmer: a large on-chip cache to which all cores are connected. The TDN network is exclusively dedicated to the implementation of the coherency protocol.

Another mechanism implemented in the Tilera architecture using the communication networks is the *TileDirect* technology that allows data received over the external interfaces to be placed directly into the tile-local memory, thus bypassing the external DDR memory and reducing memory traffic.

In our view, the scalable, low power, high bandwidth on-chip mesh interconnect as well as the mechanism that allows cache coherence on such scale (up to 100 cores) are the technologies that make the Tile processor concept unique and clearly differentiates it from other chip designs.

2.4.2.2 Graphics Processing Units

The term Graphics Processing Unit (GPU) was first used back in 1999 by NVIDIA. The introduction of the OpenGL language and Microsoft's DirectX specification resulted in more programmability of the graphics rendering process; eventually this evolution led to the introduction of a GPU architecture (by the same com-

pany, NVIDIA) that dramatically improved programmability of these chips using high level (C like) languages and turned this type of processors into an interesting choice for supercomputers targeting massively data parallel applications. NVIDIA's CUDA (Computer Unified Device Architecture) programming model was the first that enabled high level programming of GPUs; more recently the OpenCL language (which we will also cover in this book) is set to become the de facto standard for data-parallel computing in general and for programming GPUs in particular (NVIDIA itself adopted it as a layer above CUDA). On hardware side, the most radical new chip design to date by NVIDIA is the Fermi family of chips.

The Fermi architecture crams an impressive 3 billion transistors into a system with 512 CUDA cores running at 700 MHz each. A CUDA core has a pipelined 32 bit integer arithmetic logic unit (ALU) and floating point unit (FPU) with fused multiply-add support, being capable to execute an integer or floating point operation in every cycle. The CUDA cores are grouped into 16 *streaming multi-processors,* each featuring 32 CUDA cores and 16 load/store units allowing memory access operations for 16 threads per each clock cycle. There are also four special function units (SFU) that can calculate values of functions such as sin or cosine. For a schematic view of the SM architecture, see Fig. 2.7

Regarding memory, the chip supports 6 GB of ECC protected DRAM memory. For the first time, the GPU has support for cache coherency; each SM has 64 kb local memory that can be configured in a 16/48 split as cache and local store; there's also 768 kb of shared L2 cache. The chip also has a unified address space spanning both private and global memory addresses.

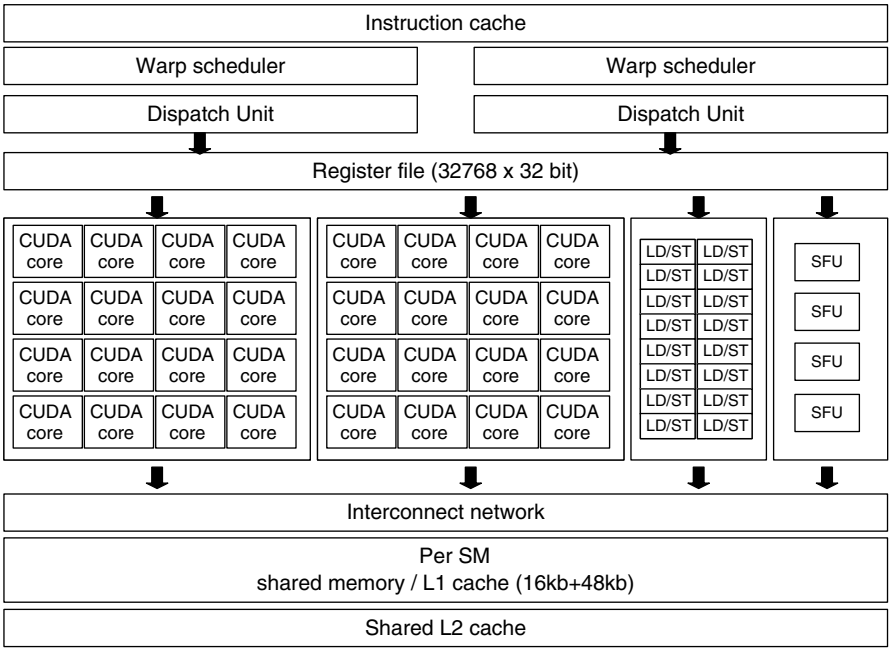


Fig. 2.7 NVIDIA streaming multiprocessor architecture

The Fermi architecture based chips have a two-level thread scheduler called GigaThread. On the chip level, an engine distributes thread blocks to different streaming multiprocessors; on each streaming multiprocessor, the so called dual warp scheduler distributes groups of 32 threads (which are called warps) to the available CUDA cores. Each streaming multiprocessor has two instruction dispatch units, thus at each cycle two warps are selected and an instruction from each warp is issued to 16 cores.

GPUs have clearly outgrown their roots in graphics processing and are now targeting the broader field of massively data-parallel applications with significant amount of scientific and floating point calculations. While a particular programming model must be followed (which we discuss later on in this book), the programmability of these chips has increased to a level that is on par with regular CPUs, turning these chips into a compelling choice for general purpose utilization.

2.4.2.3 PicoChip Architecture

We included the DSP architecture developed by picoChip because it's a prime example of successfully addressing a particular application domain with a massively multi-core processor.

The basic architecture of a picoChip DSP consists of a large number of various processing units (more than 250 in the largest configuration), connected in a mesh network (called the *picoArray*) using an interconnect resembling Tilera's iMesh technology. However, the communication is based on time division multiplexing mechanisms with the schedule decided deterministically at compile time, thus eliminating the need for execution time scheduling and arbitration. While this may be a big advantage in some cases, it will also limit the usability of the technology for more dynamic use-cases.

The processing elements in the *picoArray* can be

- *proprietary DSP cores:* 16 bit Harvard architecture processors with three-way very long instruction words, running at 160 MHz
- *hardware accelerators* for functions such as Fast Fourier Transformation (FFT), cryptography or various wireless networking algorithms that are unlikely to change and hence can be hard-coded into hardware
- *ARM core* for more complex control functions (such as operation and maintenance interface)

The DSP cores come in three variants, differentiated by target role and, as a consequence, available memory:

- *standard:* used for data-path processing with very low amount (< 1 kb) own memory
- *memory:* used for local control and buffering with approx. 8 kb of memory
- *control:* global control functions, with 64 kb of memory

Chips based on the *picoArray* architecture—due to the low amount of memory and very simple cores—have very low power consumption. For example, a model with 250 DSP cores running at 160 MHz, one ARM core at 280 MHz and several accelerators consumes just around 1 W, while capable of executing 120 billion basic arithmetic operations per second.

The picoChip architecture is a prime example of massively multi-core processors found in the embedded domain, especially telecommunication products (wireless infrastructure nodes and routers). These nodes usually perform the same, relatively simple sequence of operations on a very large amount of entities (packets, phone calls or data connections), hence such architectures are the best match in terms of balance between programmability and efficiency.

2.4.2.4 Many Integrated Core Architecture

In 2010 Intel announced their first commercial many-core chip code-named Knights Corner that will be manufactured in 22 nm and will integrate more than 50 x 86 cores. While not publicly stated, it's most likely the continuation of the Larrabee program that aimed at developing a GPU-like device.

There are few details available publicly about this device that is planned to be released in 2011. Based on an Intel presentation [3], it will be based on "vector Intel Architecture cores", which indicates an emphasis on data parallel processing. The cores will support hardware threading (at least 4 threads per core) and will share a tiled, coherent on-chip cache. The chips will likely include hardware accelerators, called *fixed function logic* in Intel's presentation. A schematic view of the architecture is shown in Fig. 2.8.

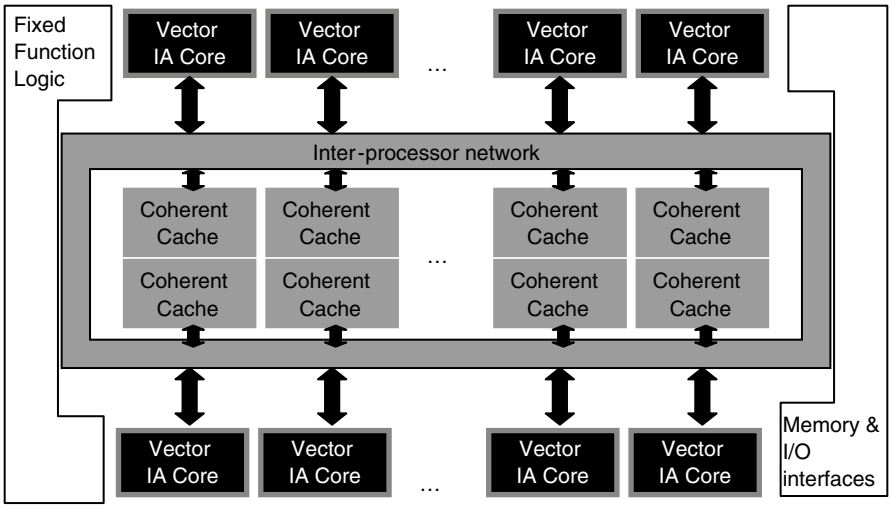


Fig. 2.8 Schematic architecture of Intel's Knights Corner chip design

In many ways this chip seems to be Intel's combined answer to the advancement of GPUs and the Tilera-type many-core architectures. Intel brands this type of chip a co-processor that a traditional server chip can use to offload parts of the application—enabled by sharing the same ISA.

2.4.3 Heterogeneous Processors

From purely hardware technology point of view, using dedicated cores for specific tasks is the best choice in terms of efficiency, as it will yield a simpler structure and less power consumption, while will deliver higher performance. As an extension, for a more complex problem, with sub-problems that require specific type of functionality, chips that combine cores with different capabilities are the optimal choice. Processors of this type are called *heterogeneous processors.*

Obviously there are other factors that make this rather simplistic line of thinking hard to argue for in full. Developing a chip has a significant cost, which is proportional to the complexity of the chip; on the other hand, the more specialized a chip is, the smallest its addressable market and hence the higher the per unit cost will be; additionally, higher complexity inherently will lead to increased software development cost. There is a trade-off point beyond which specialization cannot be anymore justified, hence the addressable market shall be sufficiently large and/or the chip design sufficiently generic so that the chip will be economically sustainable.

There are a few good examples of such chips, especially in high volume markets such as mobile computing or gaming. We will exemplify this type of processors through the Cell BE, used in game consoles as well as in supercomputers.

2.4.3.1 The Cell Broadband Engine

The Cell processor is the result of the co-operation between Sony, Toshiba and IBM and originally it was targeting gaming and other entertainment devices (including HD television sets).

The basic architecture of the Cell processor is shown in Fig. 2.9. The processor has nine cores, inter-connected through a circular bus called Element Interconnect Bus (EIB):

- one *Power Processing Element (PPE):* two-way SMT processor core running at 3.2 GHz, based on the Power Architecture ISA and equipped with 32 kb L1 and 512 kb L2 cache; it usually runs a main-stream operating system such as Linux
- eight *Synergistic Processing Elements (SPE):* DSP-like SIMD processors equipped with 256 kb of local memory (marked with LS—local store—in the figure), that can only be accessed from outside the SPE using DMA through a specialized memory flow controller unit (MFC)

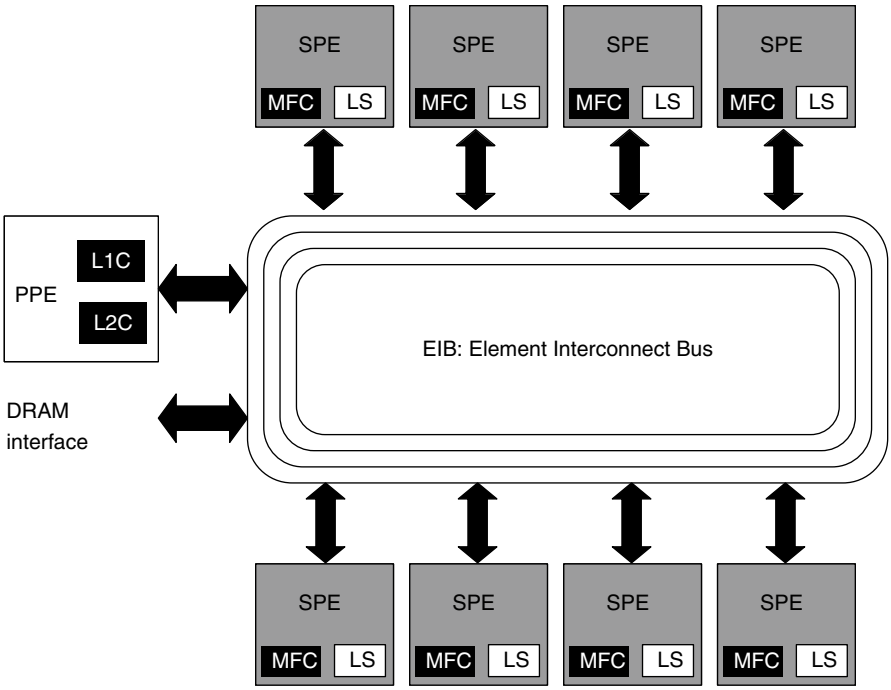


Fig. 2.9 The architecture of the Cell Broadband Engine chip

Beside the basic Cell processor, there's a special version used in supercomputers with about an 8× improvement in performance (to over 100GPFLOPS aggregated throughput).

The design of the Cell processor clearly prioritized performance over programmability: the SPEs need to be managed, scheduled and configured explicitly by the PPE, which adds significantly to the cost of software development. The division of tasks is clear: the SPEs are the number-crunching workers under the strict and necessary supervision of the PPE that takes care of all the other functions. This kind of chip architecture bolds well for game consoles (the primary users of the Cell processor), but also for compute-intensive supercomputers, where the PPE is the "visible" part of the processor capable of high-speed data crunching; internally, it can offload the work to the SPEs.

2.5 Summary

Any book on programming would be incomplete without a basic understanding of the concepts and design choices behind the actual hardware on which the software will have to execute. The goal of this chapter is to lay the (hardware) foundation

for discussing the software stack in the context of many-core programming: we surveyed the main design concepts available today, the challenges faced by chip designs when scaling up to tens or hundreds of cores as well as the emerging technologies that can help mitigate these challenges.

We believe that in the future we will see a continued increase of the amount of cores integrated on one chip—in the absence of some break-through technology, there are very few options available that can drive the performance of individual cores. This trend is already visible today: the success of GPUs and Tilera's architecture and Intel's plans for integrating more than 50 cores on a single chip all point to this direction.

The real question however is how to make use of this parallel computing power—the subject of the remaining part of this book.

References

1. Censier L M, Featrier P (1978) A New Solution to Coherence Problems in Multicache Systems," IEEE Transactions on Computers, 27(12):1112-1118
2. Gschwind M, Hofstee H P, Flachs B, Hopkin M, Watanabe Y, Yamazaki T (2006) Synergistic Processing in Cell's Multicore Architecture. IEEE Micro 26(2):10-24
3. Intel Corporation (2010) Petascale to Exascale: Extending Intel's HPC Commitment. http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf. Accessed 11 January 2011
4. Sonics MemMax Scheduler. http://www.sonicsinc.com/uploads/pdfs/memmaxscheduler_DS_021610.pdf. Accessed 10 January 2011
5. Mutlu O, Moscibroda T (2009) Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. IEEE Micro Special Issue 29(1):22-32
6. Ahn J H, Leverich J, Schreiber R S, Jouppi N P (2009) Multicore DIMM: an Energy Efficient Memory Module with Independently Controlled DRAMs. Computer Architecture Letters 8(1): 5-8
7. The OpenMP Architecture Review Board (2008) The OpenMP Application Program Interface. http://www.openmp.org/mp-documents/spec30.pdf. Accessed 10 January 2011
8. Frigo M, Leiserson C E, Randall K H (1998) The implementation of the Cilk-5 Multithreaded Language. Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation, 212-223
9. Culler D E, Gupta A, Singh J P (1998) Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann
10. Hennessy J L, Patterson D A (2006) Computer Architecture: A Quantitative Approach 4th Edition, Morgan Kaufmann
11. Wikipedia article Hyper-threading. http://en.wikipedia.org/wiki/HyperThreading. Accessed 10.1.2010
12. Mars J, Williams D, Upton D, Ghosh S, Hazelwood K (2008) A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architecture. Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms 2008, 41-50
13. Nellans D, Sudan K, Balasubramonian R, Brunvand E (2010) Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionalility. Proceedings of the 10th Workshop on Interaction between Operating Systems and Computer Architecture
14. Chua L O (1971) Memristor—the Missing Circuit Element. IEEE Transactions on Circuit Theory 18(5):507-519

15. Kurian G, Miller J E, Psota J, Eastep J, Liu J, Michel J, Kimerling L C, Agarwal A (2010) ATAC: a 1000-core Cache Coherent Processor with On-Chip Optical Network. Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques: 477-488
16. Herlihy M, Moss J E B (1993) Transactional Memory: Architectural Support for Lock-free Data Structures. Proceedings of the 20th International Symposium on Computer Architecture: 289-300
17. Falsafi B (2009) Energy-Centric Computing & Computer Architecture. Proceedings of the 2009 Workshop on New Directions in Computer Architecture
18. SPEC (2008) SPEC CPU2006. http://www.spec.org/cpu2006/. Accessed 11 January 2011
19. Åbo Akademi (2010) Cloud Software Program: SIP-Proxy and Apache Running on traditional X86 vs ARM Cortex-A9. https://research.it.abo.fi/projects/cloud/posters/POSTER_A3_Demo_2_ARM-SIP_2.pdf. Accessed 11 January 2011

http://www.springer.com/978-1-4419-9738-8

Programming Many-Core Chips
Vajda, A.
2011, XII, 228 p., Hardcover
ISBN: 978-1-4419-9738-8