

Preface

While the last 30 years has seen the computer industry driven primarily by faster and faster uniprocessors, those days have come to a close. Emerging in their place are microprocessors containing multiple processor cores that are expected to exploit parallelism. The computer industry has not abandoned uniprocessor performance as the key systems performance driver by choice, but it has been forced in this direction by technology limitations, primarily microprocessor power. Limits on transistor scaling have led to a distinct slowdown in the rate at which transistor switching energies improve. Coupled with clear indications that customers' tolerance level for power per chip has been reached, successive generations of microprocessors can now no longer significantly increase frequency. While architectural enhancements to a single-threaded core cannot pick up the slack, multicore processors promise historical performance growth rates, albeit only on sufficiently parallel applications and at the expense of additional programming effort.

Multicore design has its challenges as well. Replication of cores communicating primarily with off-chip memory and I/O leads to off-chip bandwidth demands inversely proportional to the square of the feature size, while off-chip bandwidths tend to be limited by the (more or less fixed) perimeter from which signals must escape. This trend leads to significant pressure on off-chip signaling rates, which are beginning to exceed the clock frequency of the processor core. High-bandwidth packaging such as that provided by 3D-integration may soon be required to keep up with multicore bandwidth demands.

Compared to a uniprocessor, a multicore processor is somewhat less generic, as uniprocessors are generally more efficient at emulating a parallel processor than vice versa. In the space of emerging applications with inherent concurrency, an ideal multicore processor would become part of a general-purpose system with broad enough markets to sustain the required investments in technology. However, this reality will only come to pass if the programming tools are developed and programmers are trained to deliver the intrinsic performance of these processors to the end applications.

This book is intended to provide an overview of the first generation of multicore processors and systems that are now entering the marketplace. The contributed chapters cover many topics affecting, and associated with, multicore systems including technology trends, architecture innovations, and software and

programming issues. Chapters also cover case studies of state-of-the-art commercial and research multicore systems. The case studies examine multicore implementations across different application domains, including general purpose, server, signal processing, and media/broadband. Cross-cutting themes of the book are the challenges associated with scaling multicore systems to hundreds and ultimately thousands of cores. We expect that researchers, practitioners, and students will find this book useful in their educational, research, and development endeavors.

We organize the chapters around three major themes: (1) multicore design considerations, (2) programmability innovations, and (3) case studies. While each of these areas are broader than this book can cover, together the chapters introduce the fundamentals of modern and future multicore design. While the third section of this book is dedicated to case studies, the first section also discusses prototype multicore architectures to exemplify the design considerations. In all, this book describes commercial or research implementations of seven different multicore systems or families of systems.

Multicore Design Considerations

Chapter 1 introduces tiled multicore designs, first embodied in the Raw processor chip. Tiling is both a natural and necessary method of constructing a complex system from multiple copies of a simple processor tile. Furthermore, because tiled architectures are inherently partitioned, they are less vulnerable to the deleterious effects of decreasing wire speed relative to transistor speed. Each tile also includes a network router and ports to its neighbors so that scaling a tiled design can be accomplished merely by adding more tiles, logically snapping them together at the interfaces. The perimeter of the tiled array can be used for connections to memory or to connect multiple chips together. The Raw prototype chip consists of a 4×4 array of tiles, with each tile comprising a simple processing pipeline, instruction memory, and data memory. The same design principles apply to Tiler's TILE64 chip, a commercial venture that is descended from Raw.

The key feature of the Raw system is the tight integration of the network into the processor pipeline. Raw exposes inter-tile interaction through register-mapped network ports; reading or writing the designated register name injects an operand into or extracts an operand from the network. Raw also employs a statically programmable router which executes a sequence of routing instructions to route operand packets in concert with the computation instructions executed by the core pipeline. The authors of the chapter show that applications with different characteristics can be executed by the Raw tiled substrate because the underlying computation and communication mechanisms are exposed to software. For example, exposing the operand communication enables serial programs to be partitioned across the tiles with producer/consumer instructions interacting through the operand network. The network also enables fine-grained parallel streaming computation in which small data records are processed in a pipelined fashion across multiple tiles. The Raw chip makes a compelling case for distributed architectures and for exposing

communication to software as a first-class object. These themes recur in different ways throughout the chapters of this book.

Chapter 2 focuses on the design and implementation of on-chip interconnects (networks-on-chip or NOCs), as a critical element of multicore chips. When a chip contains only a few processors and memory units, simple interconnects such as buses and small-scale crossbars suffice. However, as the demand for on-chip communication increases, so does the need for more sophisticated on-chip networks. For example, the IBM Cell chip employs a bidirectional ring network to connect its eight data processors, its Power core, and its off-chip interfaces. The Tiler chip employs a collection of mesh networks to interconnect 64 on-chip processors. This chapter first examines the fundamentals of interconnection network design, including an examination of topologies, routing, flow control, and network interfaces in the context of NOCs. The chapter also discusses the trade-offs for router microarchitecture designed to meet the area, speed, and power constraints for multicore chips.

The chapter also includes two case studies of NOCs deployed in different ways in two recent prototype chips. The TRIPS processor employs a routed network, rather than a conventional broadcast bus, to deliver operands from producing instructions to consuming instructions in a distributed uniprocessor microarchitecture and to connect the processing cores to the level-1 caches. The operand network has 25 terminals and optimizes the routers for latency with a total of one clock cycle per hop. The network is connected directly to the inputs and the outputs of the ALUs and supports fixed length, operand-sized packets. By contrast, the Intel TeraFLOPS chip connects 80 on-chip cores with a high-speed, clock-phase-skew tolerant, inter-router protocol. The TeraFLOPS routers employ a five-stage pipeline and run at 5 GHz in 65 nm. Message injection and reception is explicit with send and receive instructions that use local registers as interfaces to the network. TRIPS and the TeraFLOPS chip make different choices about routing algorithms and flow control based on the demands of their particular deployments. One message from this chapter is that NOCs are not “one-size-fits-all” and should be tailored to the needs of the system to obtain the fastest and the most efficient design. The area of NOC architecture is extremely active at present, with new innovations being published on topologies, routing algorithms, and arbitration, among other topics.

Chapter 3 examines the granularity of cores within a multicore system. A survey of commercial multicore chips shows that core size and number of cores per chip vary by almost two orders of magnitude. At one end of the spectrum are *bulldozer* processors which are large, wide issue, and currently are deployed with only 2–4 cores per chip. At the other end of the spectrum are the *termites* which are small and numerous, with as many as 128 cores per chip in 2008 technology. The trade-off between these two ends of the spectrum are driven by the application domain and the market targeted by the design. Bulldozers are the natural extensions of yesterday’s high-performance uniprocessors and target the general-purpose market. Termites typically target a narrower market in which the applications have inherent concurrency that can be more easily exploited by the more numerous yet simpler cores.

This chapter points out that the diversity in multicore granularity is an indicator of greater hardware and software specialization that steps further away from the general-purpose computing platforms developed over the last 30 years. To address this challenge, the chapter describes a general class of composable core architectures in which a chip consists of many simple cores that can be dynamically aggregated to form larger and more powerful processors. The authors describe a particular design consisting of 32 termite-sized cores which can be configured as 32 parallel uniprocessors, one 32-wide, single-threaded processor, or any number and size of processors in between. The chapter describes a set of inherently scalable microarchitectural mechanisms (instruction fetch, memory access, etc.) that are necessary to achieve this degree of flexibility and that have been prototyped and evaluated in a recent test chip. Such a flexible design would enable a single implementation to match the parallelism granularity of a range of applications and even adapt to varying granularity within a single program across its phases. Further, this class of flexible architectures may also be able to feasibly provide greater single-thread performance, a factor that is being sacrificed by many in the multicore space.

Programming for Multicores

While there are a number of challenges to the design of multicore architectures, arguably the most challenging aspect of the transition to multicore architectures is enabling mainstream application developers to make effective use of the multiple processors. To address this challenge we consider in this section the techniques of thread-level speculation (TLS) that can be used to automatically parallelize sequential applications and transactional memory (TM) which can simplify the task of writing parallel programs. As is evident from the chapters that describe these techniques, their efficient implementation requires the close interaction between systems software and hardware support.

Chapter 4 introduces speculative multithreaded or thread-level speculation architectures. The authors first provide motivation for these architectures as a solution to the problems of limited performance and implementation scalability associated with exploiting ILP using dynamic superscalar architectures and the parallel programming problems associated with traditional CMPs.

Speculatively multithreaded architectures eliminate the need for manual parallel programming by using the compiler or hardware to automatically partition a sequential program into parallel tasks. Even though these speculatively parallel tasks may have data-flow or control-flow dependencies, the hardware executes the tasks in parallel but prevents the dependencies from generating incorrect results. The authors describe in detail the approach taken to speculative multithreading in the Multiscalar architecture which was the earliest concrete example of speculative multithreading. A key element of the Multiscalar approach is the use of software and hardware to their best advantage in the implementation of speculative multithreaded execution. Software is used for task partitioning and register data-dependency tracking between tasks, which is possible with static

information. Hardware sequences the tasks at runtime and tracks inter-task memory dependencies, both of which require dynamic information. The hardware support for memory dependency tracking and memory renaming required to support speculative multithreading in the Multiscalar architecture is called the speculative version cache (SVC) and adds considerable complexity to the design of a CMP. As the authors describe, this added complexity can buy significant performance improvements for applications that cannot be parallelized with traditional auto-parallelizing technology. Despite the hardware complexity of speculative-multithreading architectures, the desire to reduce the software disruption caused by CMPs spurs continued interest in these architectures especially in conjunction with dynamic compiler technology.

Chapter 5 presents the design of Transactional Memory (TM) systems. The authors begin by describing how TM can be used to simplify parallel programming by providing a new concurrency construct that eliminates the pitfalls of using explicit locks for synchronization. While this construct is new to the mainstream parallel programming community it has been used and proven for decades in the database community. The authors describe how programming with TM can be done using the *atomic* keyword. With the high-level atomic construct, programmers have the potential to achieve the performance of a highly optimized lock-based implementation with a much simpler programming model.

The authors explain that the key implementation capabilities of a TM system include keeping multiple versions of memory and detecting memory conflicts between different transactions. These capabilities can be implemented in a software TM (STM) or in a hardware TM (HTM) using mechanisms similar to those required for speculative multithreading or a combination of both software and hardware in a hybrid-TM. Because one of the major challenges of STM implementations is performance, the chapter goes into some detail in describing compiler-based optimizations used to improve the performance of an STM by reducing the software overheads of STM operations. The key to efficient optimization is a close coupling between the compiler and the STM algorithm. While STMs support TM on current CMPs, significant performance improvements are possible by adding hardware support for TM. The chapter explains that this support can range from hardware-enhanced STM that reduces software overheads to a pure HTM that completely eliminates all software overheads and achieves the highest performance. The authors conclude that TM deployment will include the use of STMs for existing CMP hardware, but must encompass language, compiler, and runtime support to achieve adequate performance. Ultimately, hardware acceleration will be required to achieve good TM performance.

Case Studies

The last set of chapters comprise four case studies of multicore systems. In each case study, the authors describe not only the hardware design, but also discuss the demands on the systems and application software needed as a part of the multicore

ecosystem. We selected these chapters as they represent different points in the design space of multicore systems and each target different applications or multicore programming models.

Chapter 6 presents the architecture and system design of the AMD Opteron family of general-purpose multicore systems. The authors start by describing the trends and challenges for such general-purpose systems, including power consumption, memory latency, memory bandwidth, and design complexity. While multicore systems can help with some aspects, such as design complexity through replication, others are much more challenging. Power consumption must be shared by multiple processors and with strict power envelopes, not all of the processors can simultaneously operate at their peak performance. Likewise, external bandwidth (memory and interconnect) becomes a greatly constrained resource that must be shared. The challenges are no less severe on the software side, as parallel programming environments for general-purpose applications are in their infancy and system software, such as operating systems, are not yet prepared for increasing levels of concurrency of emerging multicore systems.

The authors then present the Opteron family of multicore systems, which were designed from the outset to be scaled both internally (more processors per core) and externally (more chips per system). These processor cores fit into the category of bulldozers, as they are physically large and can support high single-thread performance. The authors emphasize that because all but small-scale computer systems will be composed of many multicore chips, design for system-level scalability is critical. The chapter describes the details of the system-level architecture that is implemented on the processor chips, including multiple DRAM interfaces, inter-chip interconnection (Hypertransport), and the system-level cache hierarchy. The lessons from this chapter are that general-purpose multicore systems require a balanced design but that substantial challenges in programmability, system software, and energy efficiency still remain.

Chapter 7 details Sun Microsystems' Niagara and Niagara 2 chips, which are designed primarily for the server space in which job throughput is more important than the execution latency of any one job. The authors make the case that simple processors that lack the speculation and out-of-order execution of high-end uniprocessors are a better match for throughput-oriented workloads. Such processors with shallower pipelines and slower clock rates achieve better power efficiency and can employ multithreading as a latency tolerance technique since independent threads are abundant in these workloads. Because these processors are smaller and more power efficient, more of them can be packed onto a single chip, increasing overall throughput. These processors can be classified as *chainsaws*, as they are smaller than the bulldozers, yet larger and more full-featured than termites.

In their case studies, the authors describe in detail the microarchitecture of Sun's line of multicore chips. The first generation Niagara chip employs eight four-way multithreaded cores, for a total of 32 simultaneously executing threads. The processors share a four-banked level-2 cache, with each bank connected to an independent DRAM memory controller. The second-generation Niagara 2 doubles the number of threads per core (for a total of 64 simultaneously executing threads) and doubles the number of L2 banks to increase on-chip memory-level parallelism. While Niagara

had a single floating-point that is shared across all of the cores, each Niagara 2 core has its own floating-point and graphics unit. Both Niagara systems employ cache coherence that spans the cores on the chip and multiple chips in a system. The chapter includes experimental results that show nearly a factor of 2 boost in power efficiency (performance/Watt) over more complex core systems. Like the authors of Chapter 6, the authors of this chapter indicate that the expected increase in cores and threads will demand more system support, such as operating system virtualization.

Chapter 8 describes a family of stream processors, which are multicore systems oriented around a data-streaming execution model. Initially amenable to multimedia and scientific applications, stream processing exploits parallelism by simultaneously operating on different elements of a stream of data in a fashion similar to vector processors. By making data streams first-class objects that are explicitly moved through levels of the memory hierarchy, stream processors eliminate power and area inefficient caches. SIMD-style computation reduces the amount of control logic required for stream processors. The authors claim that these factors provide a 10–30 times improvement in power efficiency over conventional multicore architectures.

The chapter first details a stream program model which partitions programs into computation kernels and data streams. Kernels can be thought of as data filters in which an input stream or streams is transformed into an output stream. A stream program is represented as a stream flow graph which has nodes that correspond to kernel computations and edges that correspond to the streams. The program itself includes explicit operations on streams to move them through the memory hierarchy and instantiations of kernels between stream transfers. High-level programming languages such as Brook and Sequoia have been developed to target stream processing execution models.

The chapter then describes a general stream processor microarchitecture as well as three case studies: (1) Imagine stream processor, (2) Stream Processors Inc. Storm I processor, and (3) Merrimac streaming supercomputer architecture. Each of these designs shares several common design principles including an explicitly managed memory hierarchy, hardware for bulk data transfer operations, and simple lightweight arithmetic cores controlled through a hybrid SIMD/VLW execution model. These simple processing elements fall into the category of termites, since they have simple control logic and little memory per core. For example, the Storm I processor has a total of 80 ALUs organized into 16 lanes that execute in parallel. Stream processors exemplify performance and power efficiency that can be obtained when the programming model and application domain allows for explicit and organized concurrency.

Chapter 9 describes the Cell Broadband Engine architecture and its first implementations. This family of processors, jointly developed by IBM, Toshiba, and Sony falls at the upper end of the chainsaw category. The Cell Broadband Engine (Cell B.E) is a heterogeneous multiprocessor with two types of programmable cores integrated on a single chip. The Power processor element (PPE) is a Power architecture compliant core that runs the operating system and orchestrates the eight synergistic processor elements (SPEs). Per-core performance of the SPEs on compute-intensive applications is comparable to that of bulldozer cores, but each SPE requires

substantially less power, less area, and fewer transistors to implement, allowing a nine-core Cell B.E. in a chip the size of a typical dual-core bulldozer.

Like the stream processors described in Chapter 8, the SPEs achieve their efficiency by explicitly managing memory. The SPEs manage one additional level of memory than typical processors, a 256 kB local store included in each SPE. SPE DMA operations are the equivalents of the PPE's load and store operations, and access coherent shared memory on the processor in exactly the same way as the PPE's load and stores. Instead of targeting the SPE's register file, however, the DMA operations place code and data in the local store (or copy it back to main memory). The SIMD-RISC execution core of the SPE operates asynchronously on this local store. Each SPE is an autonomous single-context processor with its own program counter capable of fetching its own code and data by sending commands to the DMA unit.

The management of the local store is an added burden on the software. Initial versions of the software development environment for the Cell Broadband Engine required the application programmer to partition code and data and control their movement into and out of the local store memory. A more recent version removes the burden of code partitioning from the programmer. Following the pioneering efforts of a number of research compilers for the Cell B.E. and other multicore processors, standards-based approaches to programming this processor are now available. These programming languages and frameworks abstract the notions of locality and concurrency and can be targeted at a wide variety of multicore CPUs and (GP)GPUs allowing portable approaches to developing high-performance software and providing a path towards bringing heterogeneous computing into the mainstream.

Current and Future Directions

While the case studies in this book describe a range of architectures for different types of applications, we recognize that the design and application space is much broader. For example, we do not discuss the multicore designs of existing and emerging graphics processing units (GPUs), such as those from NVidia, Intel, and AMD, a subject which merits its own book. The graphics application domain has traditionally exposed substantial pixel and object-level parallelism, a good match to the growing number of on-chip processing elements. Interestingly, there is substantial diversity among GPUs in the number and complexity of the processing elements. The NVidia GeForce 8800 employs 128 simple cores that can operate as independent MIMD processors with non-coherent memory. Intel's Larrabee system employs fewer but larger x86-based cores with local cache and a coherent memory. AMD's Radeon HD 2900 employs four parallel stream processing units, each with 80 ALUs controlled using a hybrid of VLIW and SIMD. Successive generations of GPUs have become more programmable and currently require domain-tailored program systems such as NVidia's CUDA and AMD's CAL to achieve

both performance and programmability. The differences in approaches among the architectures targeted at the graphics domain share similarities with the distinctions between some of the architectures described in the chapters of this book.

Looking forward, Moore's law of doubling integrated circuit transistor counts every 18–24 months appears likely to continue for at least the next decade. The implication for multicore systems is the expected doubling of core count per technology generation. Given the starting point of 4–64 processors in today's commercial systems, we expect to see chips that have the capacity to contain hundreds or even thousands of cores within 10 years. However, the challenges that are described throughout this book will remain. Keeping power consumption within thermal limits will not become any easier with increasing transistor density and core counts. Programming these parallel systems is far from a solved problem, despite recent advances in parallel programming languages and tools. Developing solutions to these problems will be critical to the continued growth and performance of computer systems.

As technology matures further, eventually performance improvements will be possible only with gains in efficiency. Besides increased concurrency, the only significant means to gain efficiency appears to be increased specialization. In the context of multicore systems, cores would be specialized through configuration at run-time, at manufacture, or through unique design. Such differentiated (hybrid) multicore chips will likely be even more difficult to build and program than conventional multicore processors. However, if maturing silicon technologies lead to a lengthening of semiconductor product cycles, manufacturers will be able to afford the cost of specialization and other innovations in computer design, perhaps signaling a new golden age of computer architecture.

Acknowledgments

We would like to thank all of the contributing authors for their hard work in producing a set of excellent chapters. They all responded well to editorial feedback intended to provide a level of cohesion to the independently authored chapters. We would also like to thank Anantha Chandrakasan, Chief Editor of Springer's Series on Integrated Circuits and Systems, for recruiting us to assemble this book. Thanks go to the editorial staff at Springer, in particular Katelyn Stanne and Carl Harris for their patience and support of the production of this book.

Texas, USA
California, USA
Texas, USA

Stephen W. Keckler
Kunle Olukotun
H. Peter Hofstee

Multicore Processors and Systems

Keckler, S.W.; Kunle, O.; Hofstee, H.P. (Eds.)

2009, XVIII, 301 p., Hardcover

ISBN: 978-1-4419-0262-7