Integrated Circuits and Systems

Series Editor
A. Chandrakasan
Massachusetts Institute of Technology
Cambridge, Massachusetts

For further volumes:
http://www.springer.com/series/7236
Stephen W. Keckler · Kunle Olukotun ·
H. Peter Hofstee
Editors

Multicore Processors
and Systems

Springer
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 unprocessors 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 Tilera’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 Tilera 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/VLIW 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
Stephen W. Keckler
California, USA
Kunle Olukotun
Texas, USA
H. Peter Hofstee
# Contents

1 Tiled Multicore Processors ........................................... 1  
   Michael B. Taylor, Walter Lee, Jason E. Miller, David Wentzlaff,  
   Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul R. Johnson,  
   Jason S. Kim, James Psota, Arvind Saraf, Nathan Shnidman,  
   Volker Strumpen, Matthew I. Frank, Saman Amarasinghe, and  
   Anant Agarwal

2 On-Chip Networks for Multicore Systems .......................... 35  
   Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal

3 Composable Multicore Chips ..................................... 73  
   Doug Burger, Stephen W. Keckler, and Simha Sethumadhavan

4 Speculatively Multithreaded Architectures ....................... 111  
   Gurindar S. Sohi and T.N. Vijaykumar

5 Optimizing Memory Transactions for Multicore Systems ............ 145  
   Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha

6 General-Purpose Multi-core Processors ............................ 173  
   Chuck Moore and Pat Conway

7 Throughput-Oriented Multicore Processors .......................... 205  
   James Laudon, Robert Golla, and Greg Grohoski

8 Stream Processors ...................................................... 231  
   Mattan Erez and William J. Dally

9 Heterogeneous Multi-core Processors: The Cell  
   Broadband Engine .............................................................. 271  
   H. Peter Hofstee

Index ................................................................. 297
Contributors

Ali-Reza Adl-Tabatabai  Intel Corporation, Hillsboro, OR USA
Anant Agarwal  MIT CSAIL, Cambridge, MA USA
Saman Amarasinghe  MIT CSAIL, Cambridge, MA USA
Ian Bratt  Tilera Corporation, Westborough, MA USA
Doug Burger  The University of Texas at Austin, Austin, TX USA
Pat Conway  Advanced Micro Devices Inc., Sunnyvale, CA USA
William J. Dally  Stanford University, Stanford, CA USA
Mattan Erez  The University of Texas at Austin, Austin, TX USA
Matthew I. Frank  University of Illinois at Urbana – Champaign, Urbana, IL USA
Robert Golla  Sun Microsystems, Austin, TX USA
Ben Greenwald  Veracode, Burlington, MA USA
Greg Grohoski  Sun Microsystems, Austin, TX USA
Henry Hoffmann  MIT CSAIL, Cambridge, MA USA
H. Peter Hofstee  IBM Systems and Technology Group, Austin, TX USA
Paul R. Johnson  MIT CSAIL, Cambridge, MA USA
Stephen W. Keckler  The University of Texas at Austin, Austin, TX USA
Jason S. Kim  MIT CSAIL, Cambridge, MA USA
Christos Kozyrakis  Stanford University, Stanford, CA USA
James Laudon  Google, Madison, WI USA
Walter Lee  Tilera Corporation, Westborough, MA USA
Jason E. Miller  MIT CSAIL, Cambridge, MA USA
Chuck Moore  Advanced Micro Devices Inc., Sunnyvale, CA USA
Li-Shiuan Peh  Princeton University, Princeton, New Jersey, USA
James Psota  MIT CSAIL, Cambridge, MA USA
Bratin Saha  Intel Corporation, Hillsboro, OR USA
Arvind Saraf  Swasth Foundation, Bangalore India
Simha Sethumadhavan  Columbia University, New York, NY USA
Nathan Shnidman  The MITRE Corporation, Bedford, MA USA
Gurindar S. Sohi  University of Wisconsin, Madison, WI USA
Volker Strumpen  IBM Austin Research Laboratory, Austin, TX USA
Michael B. Taylor  University of California, San Diego, La Jolla, CA USA
Sriram Vangal  Intel Corporation, Hillsboro, OR USA
T.N. Vijaykumar  Purdue University, West Lafayette, IN USA
David Wentzlaff  MIT CSAIL, Cambridge, MA USA
Chapter 1
Tiled Multicore Processors

Michael B. Taylor, Walter Lee, Jason E. Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jason S. Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman Amarasinghe, and Anant Agarwal

Abstract For the last few decades Moore’s Law has continually provided exponential growth in the number of transistors on a single chip. This chapter describes a class of architectures, called tiled multicore architectures, that are designed to exploit massive quantities of on-chip resources in an efficient, scalable manner. Tiled multicore architectures combine each processor core with a switch to create a modular element called a tile. Tiles are replicated on a chip as needed to create multicores with any number of tiles. The Raw processor, a pioneering example of a tiled multicore processor, is examined in detail to explain the philosophy, design, and strengths of such architectures. Raw addresses the challenge of building a general-purpose architecture that performs well on a larger class of stream and embedded computing applications than existing microprocessors, while still running existing ILP-based sequential programs with reasonable performance. Central to achieving this goal is Raw’s ability to exploit all forms of parallelism, including ILP, DLP, TLP, and Stream parallelism. Raw approaches this challenge by implementing plenty of on-chip resources – including logic, wires, and pins – in a tiled arrangement, and exposing them through a new ISA, so that the software can take advantage of these resources for parallel applications. Compared to a traditional superscalar processor, Raw performs within a factor of 2x for sequential applications with a very low degree of ILP, about 2x–9x better for higher levels of ILP, and 10x–100x better when highly parallel applications are coded in a stream language or optimized by hand.

J.E. Miller (✉)
MIT CSAIL 32 Vassar St, Cambridge, MA 02139, USA
e-mail: jasonm@alum.mit.edu

1.1 Introduction

For the last few decades, Moore’s Law has continually provided exponential growth in the number of transistors on a single chip. The challenge for computer architects is to find a way to use these additional transistors to increase application performance. This chapter describes a class of architectures, called tiled multicore architectures, that are designed to exploit massive quantities of on-chip resources in an efficient and scalable manner. Tiled multicore architectures combine each processor core with a switch to create a modular element called a tile. Tiles are replicated on a chip as needed to create multicores with a few or large numbers of tiles.

The Raw processor, a pioneering example of a tiled multicore processor, will be examined in detail to explain the philosophy, design, and strengths of such architectures. Raw addresses the challenge of building a general-purpose architecture that performs well on a larger class of stream and embedded computing applications than existing microprocessors, while still running existing ILP-based sequential programs with reasonable performance. Central to achieving this goal is Raw’s ability to exploit all forms of parallelism, including instruction-level parallelism (ILP), data-level parallelism (DLP), task or thread-level parallelism (TLP), and stream parallelism.

1.1.1 The End of Monolithic Processors

Over the past few decades, general-purpose processor designs have evolved by attempting to automatically find and exploit increasing amounts of parallelism in sequential programs: first came pipelined single-issue processors, then in-order superscalars, and finally out-of-order superscalars. Each generation has employed larger and more complex circuits (e.g., highly ported register files, huge bypass networks, reorder buffers, complex cache hierarchies, and load/store queues) to extract additional parallelism from a simple single-threaded program.

As clock frequencies have increased, wire delay within these large centralized structures has begun to limit scalability [9, 28, 2]. With a higher clock frequency, the fraction of a chip that a signal can reach in a single clock period becomes smaller. This makes it very difficult and costly to scale centralized structures to the sizes needed by large monolithic processors. As an example, the Itanium II processor [32] spends over half of its critical path in the bypass paths of the ALUs (which form a large centralized interconnect). Techniques like resource partitioning and super-pipelining attempt to hide the realities of wire delay from the programmer, but create other inefficiencies that result in diminishing performance returns.

Besides the performance implications, many of these large centralized structures are power-hungry and very costly to design and verify. Structures such as bypass networks and multiported register files grow with the square or cube of the issue-width in monolithic superscalars. Because power relates to both area and frequency in CMOS VLSI design, the power consumed by these complex monolithic
processors is approaching VLSI limits. Increased complexity also results in increased design and verification costs. For large superscalar processors, verification can account for as much as 70% of total development cost [10].

Due to these limited performance improvements, skyrocketing energy consumption, and run-away design costs, it has become clear that large monolithic processor architectures will not scale into the billion-transistor era [35, 2, 16, 36, 44].

1.1.2 Tiled Multicore Architectures

Tiled multicore architectures avoid the inefficiencies of large monolithic sequential processors and provide unlimited scalability as Moore’s law provides additional transistors per chip. Like all multicore processors, tiled multicores (such as Raw, TRIPS [37], Tilera’s TILE64 [50, 7], and Intel’s Tera-Scale Research Processor [6]) contain several small computational cores rather than a single large one. Since simpler cores are more area- and energy-efficient than larger ones, more functional units can be supported within a single chip’s area and power budget [1]. Specifically, in the absence of other bottlenecks, multicores increase throughput in proportion to the number of cores for parallel workloads without the need to increase clock frequency. The multicore approach is power efficient because increasing the clock frequency requires operating at proportionally higher voltages in optimized processor designs, which can increase power by the cube of the increase in frequency. However, the key concept in tiled multicores is the way in which the processing cores are interconnected. In a tiled multicore, each core is combined with a communication network router, as shown in Fig. 1.1, to form an independent modular “tile.” By replicating

fig. 1.1 A tiled multicore processor is composed of an array of tiles. Each tile contains an independent compute core and a communication switch to connect it to its neighbors. Because cores are only connected through the routers, this design can easily be scaled by adding additional tiles
tiles across the area of a chip and connecting neighboring routers together, a complete on-chip communication network is created.

The use of a general network router in each tile distinguishes tiled multicore from other mainstream multicore such as Intel’s Core processors, Sun’s Niagara [21], and the Cell Broadband Engine [18]. Most of these multicore have distributed processing elements but still connect cores together using non-scalable centralized structures such as bus interconnects, crossbars, and shared caches. The Cell processor uses ring networks that are physically scalable but can suffer from significant performance degradation due to congestion as the number of cores increases. Although these designs are adequate for small numbers of cores, they will not scale to the thousand-core chips we will see within the next decade.

Tiled multicore distribute both computation and communication structures providing advantages in efficiency, scalability, design costs, and versatility. As mentioned previously, smaller simpler cores are faster and more efficient due to the scaling properties of certain internal processor structures. In addition, they provide fast, cheap access to local resources (such as caches) and incur extra cost only when additional distant resources are required. Centralized designs, on the other hand, force every access to incur the costs of using a single large, distant resource. This is true to a lesser extent even for other multicore designs with centralized interconnects. Every access that leaves a core must use the single large interconnect. In a tiled multicore, an external access is routed through the on-chip network and uses only the network segments between the source and destination.

Tiled multicore architectures are specifically designed to scale easily as improvements in process technology provide more transistors on each chip. Because tiled multicore use distributed communication structures as well as distributed computation, processors of any size can be built by simply laying down additional tiles. Moving to a new process generation does not require any redesign or re-verification of the tile design. Besides future scalability, this property has enormous advantages for design costs today. To design a huge billion-transistor chip, one only needs to design, layout, and verify a small, relatively simple tile and then replicate it as needed to fill the die area. Multicores with centralized interconnect allow much of the core design to be re-used, but still require some customized layout for each core. In addition, the interconnect may need to be completely redesigned to add additional cores.

As we will see in Sect. 1.5, tiled multicore are also much more versatile than traditional general-purpose processors. This versatility stems from the fact that, much like FPGAs, tiled multicore provide large quantities of general processing resources and allow the application to decide how best to use them. This is in contrast to large monolithic processors where the majority of die area is consumed by special-purpose structures that may not be needed by all applications. If an application does need a complex function, it can dedicate some of the resources to emulating it in software. Thus, tiled multicore are, in a sense, more general than general-purpose processors. They can provide competitive performance on single-threaded ILP (instruction-level parallelism) applications as well as applications that
are traditionally the domain of DSPs, FPGAs, and ASICs. As demonstrated in Sect. 1.4, they do so by supporting multiple models of computation such as ILP, streaming, TLP (task or thread-level parallelism), and DLP (data-level parallelism).

### 1.1.3 Raw: A Prototype Tiled Multicore Processor

The Raw processor is a prototype tiled multicore. Developed in the Computer Architecture Group at MIT from 1997 to 2002, it is one of the first multicore processors. The design was initially motivated by the increasing importance of managing wire delay and the desire to expand the domain of “general-purpose” processors into the realm of applications traditionally implemented in ASICs. To obtain some intuition on how to approach this challenge, we conducted an early study [5, 48] on the factors responsible for the significantly better performance of application-specific VLSI chips. We identified four main factors: specialization; exploitation of parallel resources (gates, wires, and pins); management of wires and wire delay; and management of pins.

1. **Specialization:** ASICs specialize each “operation” at the gate level. In both the VLSI circuit and microprocessor context, an operation roughly corresponds to the unit of work that can be done in one cycle. A VLSI circuit forms operations by combinational logic paths, or “operators,” between flip-flops. A microprocessor, on the other hand, has an instruction set that defines the operations that can be performed. Specialized operators, for example, for implementing an incompatible floating-point operation, or implementing a linear feedback shift register, can yield an order of magnitude performance improvement over an extant general-purpose processor that may require many instructions to perform the same one-cycle operation as the VLSI hardware.

2. **Exploitation of Parallel Resources:** ASICs further exploit plentiful silicon area to implement enough operators and communications channels to sustain a tremendous number of parallel operations in each clock cycle. Applications that merit direct digital VLSI circuit implementations typically exhibit massive, operation-level parallelism. While an aggressive VLIW implementation like Intel’s Itanium II [32] executes six instructions per cycle, graphics accelerators may perform hundreds or thousands of word-level operations per cycle. Because they operate on very small word operands, logic emulation circuits such as Xilinx II Pro FPGAs can perform hundreds of thousands of operations each cycle. Clearly the presence of many physical execution units is a minimum prerequisite to the exploitation of the same massive parallelism that ASICs are able to exploit.

---

1 However, the term “multicore” was not coined until more recently when commercial processors with multiple processing cores began to appear in the marketplace.
3. **Management of Wires and Wire Delay**: ASIC designers can place and wire communicating operations in ways that minimize wire delay, minimize latency, and maximize bandwidth. ASIC designers manage wire delay inherent in large distributed arrays of function units in multiple steps. First, they place close together operations that need to communicate frequently. Second, when high bandwidth is needed, they create multiple customized communication channels. Finally, they introduce pipeline registers between distant operators, thereby converting propagation delay into pipeline latency. By doing so, the designer acknowledges the inherent trade-off between parallelism and latency: leveraging more resources requires signals to travel greater distances.

4. **Management of Pins**: ASICs customize the usage of their pins. Rather than being bottlenecked by a cache-oriented multi-level hierarchical memory system (and subsequently by a generic PCI-style I/O system), ASICs utilize their pins in ways that fit the applications at hand, maximizing realizable I/O bandwidth or minimizing latency. This efficiency applies not just when an ASIC accesses external DRAMs, but also in the way that it connects to high-bandwidth input devices like wide-word analog-to-digital converters, CCDs, and sensor arrays. There are currently few easy ways to arrange for these devices to stream data into a general-purpose microprocessor in a high-bandwidth way, especially since DRAM must almost always be used as an intermediate buffer.

The goal of the Raw project is to build a general-purpose microprocessor that can leverage the above four factors while still running existing ILP-based sequential applications with reasonable performance. In addition, the design must be scalable in the face of ever-increasing wire delays. It needs to implement the gamut of general-purpose features that we expect in a microprocessor such as functional unit virtualization, unpredictable interrupts, instruction virtualization, data caching, and context switching. To achieve good performance it also needs to exploit ILP in sequential programs and allow multiple threads of control to communicate and coordinate efficiently. Raw takes the following approach to leveraging the four factors behind the success of ASICs:

1. Raw implements the most common operations needed by ILP, stream, or TLP applications in specialized hardware mechanisms. Most of the primitive mechanisms are exposed to software through a new ISA. These mechanisms include the usual integer and floating-point operations; specialized bit manipulation ops; scalar operand routing between adjacent tiles; operand bypass between functional units, registers and I/O queues; and data cache access (e.g., load with tag check).

2. Raw implements a large number of these operators which exploit the copious VLSI resources—including gates, wires, and pins—and exposes them through a new ISA, such that the software can take advantage of them for both ILP and highly parallel applications.

3. Raw manages the effect of wire delays by exposing the wiring channel operators to the software, so that the software can account for latencies by orchestrating
both scalar and stream data transport. By orchestrating operand flow on the interconnect, Raw can also create customized communications patterns. Taken together, the wiring channel operators provide the abstraction of a scalar operand network [44] that offers very low latency for scalar data transport and enables the exploitation of ILP.

4. Raw software manages the pins for cache data fetches and for specialized stream interfaces to DRAM or I/O devices.

1.1.4 Chapter Overview

The rest of this chapter describes the Raw processor in more detail and evaluates the extent to which it succeeds in serving as a more versatile general-purpose processor. Section 1.2 provides an overview of the Raw architecture and its mechanisms for specialization, exploitation of parallel resources, orchestration of wires, and management of pins. Section 1.3 describes the specific implementation of the prototype Raw chip. Section 1.4 evaluates the performance of Raw on applications drawn from several classes of computation including ILP, streams and vectors, server, and bit-level embedded computation. Section 1.5 introduces a new metric called “versatility” that folds into a single scalar number the performance of an architecture over many application classes. Section 1.6 follows with a detailed discussion of other related processor architectures. Finally, Section 1.7 concludes with a summary of our findings.

1.2 Raw Architecture Overview

The Raw architecture supports an ISA that provides a parallel interface to the gate, pin, and wiring resources of the chip through suitable high-level abstractions. As illustrated in Fig. 1.2, the Raw processor exposes the copious gate resources of the

![Fig. 1.2](image-url)
chip by dividing the usable silicon area into an array of 16 identical, programmable tiles. Each tile contains a processing core and communication routers to connect it to neighboring tiles. The processing core contains an eight-stage in-order single-issue MIPS-style processing pipeline (right side of Fig. 1.2), a four-stage single-precision pipelined FPU, a 32 KB data cache, and a 32 KB software-managed instruction cache [30] for the processing pipeline. There are two types of communication routers (static and dynamic), with a 64 KB software-managed instruction cache for the static router. Each tile is sized so that the amount of time for a signal to travel through a small amount of logic and across the tile is one clock cycle. Future Raw-like processors will likely have hundreds or even thousands of tiles.

1.2.1 On-Chip Networks

The tiles are interconnected by four 32-bit full-duplex on-chip networks, consisting of over 12,500 wires (see Fig. 1.2). Two of the networks are static (routes are specified at compile time) and two are dynamic (routes are specified at run time). Each tile is connected only to its four neighbors. Every wire is registered at the input to its destination tile. This means that the longest wire in the system is no longer than the length or width of a tile. This property ensures high clock speeds, and the continued scalability of the architecture.

The design of Raw's on-chip interconnect and its interface with the processing pipeline are its key innovative features. These on-chip networks are exposed to the software through the Raw ISA, thereby giving the programmer or compiler the ability to directly program the wiring resources of the processor, and to carefully orchestrate the transfer of data values between the computational portions of the tiles—much like the routing in an ASIC. Effectively, the wire delay is exposed to the user as network hops. A route between opposite corners of the processor takes six hops, which corresponds to approximately six cycles of wire delay. To minimize the latency of inter-tile scalar data transport (which is critical for ILP) the on-chip networks are not only register-mapped but also integrated directly into the bypass paths of the processor pipeline. The register-mapped ports allow an instruction to place a value on the network with no overhead. Similarly, instructions using values from the network simply read from the appropriate register. The programmable switches bear the responsibility of routing operands through the network.

Raw’s on-chip interconnects belong to the class of scalar operand networks [44], which provide an interesting way of looking at modern day processors. The register file used to be the central communication mechanism between functional units in a processor. Starting with the first pipelined processors, the bypass network has become largely responsible for the communication of active values, and the register file is more of a checkpointing facility for inactive values. The Raw networks (in particular the static networks) serve as 2-D bypass networks, bridging between the bypass networks of separate cores.
The static router in each tile contains a 64 KB software-managed instruction cache and a pair of routing crossbars. Compiler-generated routing instructions are 64 bits and encode a small command (e.g., conditional branch with/without decrement) and several routes, one for each crossbar output. Each Raw static router, also known as a switch processor, contains a four-way crossbar, with each way corresponding to one of the cardinal directions (north, east, south, and west). The single-cycle routing instructions are one example of Raw’s use of specialization. Because the router program memory is cached, there is no practical architectural limit on the number of simultaneous communication patterns that can be supported in a computation. This feature, coupled with the extremely low latency and low occupancy of the in-order inter-tile ALU-to-ALU operand delivery (three cycles nearest neighbor) distinguishes Raw from prior systolic or message passing systems [4, 14, 24].

Table 1.1 shows the breakdown of costs associated with sending a single-word message on Raw’s static network. Because the network interfaces are register-mapped, the send occupancy is zero—the instruction that produces the result to be sent can directly place it on the network without requiring any additional instructions. Similarly, the receiving instruction can pull its input value directly off of the network. If the switch processor is already waiting for the next value to come from the ALU, that value will be placed on the network immediately. It then takes one cycle for each tile it passes through on the way to its destination. At the receiving end, it takes two cycles to pull the value off the network and queue it up for the ALU.

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending processor occupancy</td>
<td>0</td>
</tr>
<tr>
<td>Latency from ALU output to network</td>
<td>0</td>
</tr>
<tr>
<td>Latency per hop</td>
<td>1</td>
</tr>
<tr>
<td>Latency from network to ALU input</td>
<td>2</td>
</tr>
<tr>
<td>Receiving processor occupancy</td>
<td>0</td>
</tr>
</tbody>
</table>

Raw’s two dynamic networks support cache misses, interrupts, dynamic messages, and other asynchronous events. The two networks use dimension-ordered routing and are structurally identical. One network, the memory network, follows a deadlock-avoidance strategy to avoid end-point deadlock. It is used in a restricted manner by trusted clients such as data caches, DMA, and I/O. The second network, the general network, is used by untrusted clients, and relies on a deadlock recovery strategy [24].

Raw fully supports context switches. On a context switch, the contents of the processor registers and the general and static networks on a subset of the Raw chip occupied by the process (possibly including multiple tiles) are saved off. The process and its network data can then be restored to the same position or a new offset on the Raw grid.
1.2.2 Direct I/O Interfaces

On the edges of the network, the network channels are multiplexed down onto the pins of the chip to form flexible I/O ports that can be used for DRAM accesses or external device I/O. To toggle a pin, the user programs one of the on-chip networks to route a value off the side of the array. The 1657-pin CCGA (ceramic column-grid array) package used on Raw provides 14 full-duplex, 32-bit I/O ports. Raw implementations with fewer pins are made possible via logical channels (as is already the case for two out of the 16 logical ports), or simply by bonding out only a subset of the ports.

The static and dynamics networks, the data cache of the compute processors, and the external DRAMs connected to the I/O ports comprise Raw’s memory system. The memory network is used for cache-based memory traffic while the static and general dynamic networks are used for stream-based memory traffic. Systems designed for memory-intensive applications can have up to 14 full-duplex full-bandwidth DRAM banks by placing one on each of the chip’s 14 physical I/O ports. Minimal embedded Raw systems may eliminate DRAM altogether: booting from a single ROM and executing programs entirely out of the on-chip memories. In addition to transferring data directly to the tiles, off-chip devices connected to the I/O ports can route data through the on-chip networks to other devices in order to perform glueless DMA and peer-to-peer communication.

1.2.3 ISA Analogs to Physical Resources

By creating first class architectural analogs to the physical chip resources, Raw attempts to minimize the ISA gap—that is, the gap between the resources that a VLSI chip has available and the amount of resources that are usable by software. Unlike conventional ISAs, Raw exposes the quantity of all three underlying physical resources (gates, wires, and pins) in the ISA. Furthermore, it does this in a manner that is backwards-compatible—the instruction set does not change with varying degrees of resources.

Table 1.2 contrasts the ways that the Raw ISA and conventional ISAs expose physical resources to the programmer. Because the Raw ISA has more direct interfaces, Raw processors can have more functional units and more flexible and efficient pin utilization. High-end Raw processors will typically have more pins, because the architecture is better at turning pin count into performance and functionality.

<table>
<thead>
<tr>
<th>Physical entity</th>
<th>Raw ISA analog</th>
<th>Conventional ISA analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gates</td>
<td>Tiles, new instructions</td>
<td>New instructions</td>
</tr>
<tr>
<td>Wires, Wire delay</td>
<td>Routes, Network hops</td>
<td>None</td>
</tr>
<tr>
<td>Pins</td>
<td>I/O ports</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 1.2 How Raw converts increasing quantities of physical entities into ISA entities
Finally, Raw processors are more predictable and have higher frequencies because of the explicit exposure of wire delay.

This approach makes Raw scalable. Creating subsequent, more powerful generations of the processor is straightforward and is as simple as stamping out as many tiles and I/O ports as the silicon die and package allow. The design has no centralized resources, no global buses, and no structures that get larger as the tile or pin count increases. Finally, the longest wire, the design complexity, and the verification complexity are all independent of transistor count.

### 1.3 Raw Chip Implementation

The Raw chip is a 16-tile prototype implemented in IBM’s 180 nm, 1.8 V, six-layer, CMOS 7SF SA-27E copper ASIC process. A die photograph of the Raw chip is shown in Fig. 1.3. One hundred and twenty chips were received from IBM in October 2002. Although the Raw array is only $16 \times 16$ mm, an $18.2 \times 18.2$ mm die is used to allow for the high pin-count package. The 1657-pin ceramic column grid array (CCGA) package provides 1080 high-speed transceiver logic (HSTL) I/O pins. Measurements indicate that the chip core averages 18.2 watts at 425 MHz (with unused functional units and memories quiesced and unused data I/O pins tri-stated). The target clock frequency was 225 MHz under worst-case conditions, which is competitive with other 180 nm lithography ASIC processors, such as VIRAM [22], Imagine [19], and Tensilica’s Xtensa series. The nominal running frequency is typically higher—the Raw chip core, running at room temperature, reaches 425 MHz at 1.8 V, and 500 MHz at 2.2 V. This compares favorably to IBM-implemented microprocessors in the same process: the PowerPC 405 GP runs at 266–400 MHz, while the follow-on PowerPC 440 GP reaches 400–500 MHz.

![Die photo of the Raw chip (left) and photo of the Raw prototype motherboard (right)](image)
The processor is aggressively pipelined, with conservative treatment of the control paths in order to ensure that only reasonable efforts would be required to close timing in the backend. Despite these efforts, wire delay inside a tile was still large enough to warrant the creation of a special infrastructure to place the cells in the timing and congestion-critical datapaths. More details on the Raw implementation are available in [43].

As one can infer from Sect. 1.5, moving from a single-issue compute processor to a dual-issue compute processor would have likely improved performance on low-ILP applications. Estimates indicate that such a compute processor would have easily fit in the remaining empty space within a tile. The frequency impact of transitioning from single-issue to dual-issue is generally held to be small.

A prototype motherboard (shown in Fig. 1.3) using the Raw chip was designed in collaboration with the Information Sciences Institute (ISI) East. It includes a single Raw chip, SDRAM chips, I/O interfaces, and interface FPGAs. A larger system consisting of separate processor and I/O boards has also been developed with ISI. Each processor board contains four Raw chips connected in a $2 \times 2$ array. Multiple processor boards can be connected to form systems consisting of up to 64 Raw chips, thereby forming virtual Raw processors with up to 1,024 tiles.

### 1.4 Results

This section presents measurement and experimental results of the Raw microprocessor. We begin by explaining our experimental methodology. Then we present some basic hardware statistics. The remainder of the section focuses on evaluating how well Raw supports a range of programming models and application types. The domains we examine include ILP computation, stream and embedded computation, TLP server workloads, and bit-level computation. The performance of Raw in these individual areas are presented as comparison to a reference 600 MHz Pentium III.

As you will see, in some cases Raw achieves greater than $16 \times$ speedup (either vs. the Pentium III or vs. a single tile) even though it contains (at most) 16 times as many functional units. This super-linear speedup can be attributed to compounding or additive effects from several different factors. These factors are listed in Table 1.3 along with the maximum possible speedup that each can enable and are briefly discussed here.

<table>
<thead>
<tr>
<th>Factor responsible</th>
<th>Max. Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tile parallelism (exploitation of gates)</td>
<td>$16 \times$</td>
</tr>
<tr>
<td>Load/store elimination (management of wires)</td>
<td>$4 \times$</td>
</tr>
<tr>
<td>Streaming mode vs. cache thrashing (management of wires)</td>
<td>$15 \times$</td>
</tr>
<tr>
<td>Streaming I/O bandwidth (management of pins)</td>
<td>$60 \times$</td>
</tr>
<tr>
<td>Increased cache/register size (exploitation of gates)</td>
<td>$\sim 2 \times$</td>
</tr>
<tr>
<td>Bit manipulation instructions (specialization)</td>
<td>$3 \times$</td>
</tr>
</tbody>
</table>
1. When all 16 tiles can be used, the speedup can be 16-fold.

2. If $a$, $b$, and $c$ are variables in memory, then an operation of the form $c = a + b$ in a traditional load–store RISC architecture will require a minimum of four instructions—two loads, one add, and one store. Stream architectures such as Raw can accomplish the operation with a single instruction (for a speedup of $4 \times$) by eliminating the loads and stores. For long streams of data this is done by issuing bulk data stream requests to the memory controller and then processing data directly from the network without loading and storing it in the cache. This effect also works on a smaller scale if an intermediate value is passed directly over the network from a neighboring tile instead of being passed through memory.

3. When vector lengths exceed the cache size, streaming data from off-chip DRAM directly into the ALU achieves $7.5 \times$ the throughput of cache accesses (each cache miss transports eight words in 60 cycles, while streaming can achieve one word per cycle). The streaming effect is even more powerful with strided requests that use only part of a full cache line. In this case, streaming throughput is 15 times greater than going through the cache.

4. Raw has $60 \times$ the I/O bandwidth of the Pentium III. Furthermore, Raw’s direct programmatic interface to the pins enables more efficient utilization.

5. When multiple tiles are used in a computation, the effective number of registers and cache lines is increased, allowing a greater working set to be accessed without penalty. We approximate the potential speedup from this effect as two-fold.

6. Finally, specialized bit manipulation instructions can optimize table lookups, shifts, and logical operations. We estimate the potential speedup from these instructions as three-fold.

### 1.4.1 Experimental Methodology

The evaluation presented here makes use of a validated cycle-accurate simulator of the Raw chip. Using the validated simulator as opposed to actual hardware facilitates the normalization of differences with a reference system, e.g., DRAM memory latency, and instruction cache configuration. It also allows for exploration of alternative motherboard configurations. The simulator was meticulously verified against the gate-level RTL netlist to have exactly the same timing and data values for all 200,000 lines of the hand-written assembly test suite, as well as for a number of C applications and randomly generated tests. Every stall signal, register file write, SRAM write, on-chip network wire, cache state machine transition, interrupt signal, and chip signal pin matches in value on every cycle between the two. This gate-level RTL netlist was then shipped to IBM for manufacturing. Upon receipt of the chip, a subset of the tests was compared to the actual hardware to verify that the chip was manufactured according to specifications.

**Reference Processor:** To have a meaningful evaluation of a new architecture, it is important to compare the new empirical data to an existing commercial processor.
For this evaluation, a 600 MHz Pentium III (P3) was selected as the reference processor. The 600 MHz P3, is implemented in the same process generation as the Raw chip (180 nm) and represents the middle of the initial production frequency range, before extensive process or speedpath tuning. This levels the playing field somewhat when comparing the P3 to the “first-silicon” prototype Raw chip. In addition, the P3’s functional unit latencies and level of pipelining are nearly identical to Raw, making direct cycle-count comparisons meaningful. Table 1.4 lists a few key characteristics of Raw and the P3 for comparison. For a more detailed comparison and discussion of why the 600 MHz P3 is an appropriate reference processor see [46].

**Table 1.4** Comparison of processor characteristics for Raw and P3-Coppermine

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Raw (IBM ASIC)</th>
<th>Pentium III (Intel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lithography generation</td>
<td>180 nm</td>
<td>180 nm</td>
</tr>
<tr>
<td>Metal layers</td>
<td>6 Cu</td>
<td>6 Al</td>
</tr>
<tr>
<td>Frequency</td>
<td>425 MHz</td>
<td>600 MHz</td>
</tr>
<tr>
<td>Die area(^{a})</td>
<td>331 mm(^{2})</td>
<td>106 mm(^{2})</td>
</tr>
<tr>
<td>Signal pins</td>
<td>(~ 1100)</td>
<td>(~ 190)</td>
</tr>
<tr>
<td>Sustained issue width</td>
<td>1 in-order (per tile)</td>
<td>3 out-of-order</td>
</tr>
<tr>
<td>ALU latency/occupancy</td>
<td>1 / 1</td>
<td>1 / 1</td>
</tr>
<tr>
<td>Load latency/occupancy (hit)</td>
<td>3 / 1</td>
<td>3 / 1</td>
</tr>
<tr>
<td>Store latency/occupancy (hit)</td>
<td>(~ / 1)</td>
<td>(~ / 1)</td>
</tr>
<tr>
<td>Mispredict penalty</td>
<td>3</td>
<td>10–15</td>
</tr>
<tr>
<td>L1 data cache size</td>
<td>32 K</td>
<td>16 K</td>
</tr>
<tr>
<td>L1 instruction cache size</td>
<td>32 K</td>
<td>16 K</td>
</tr>
<tr>
<td>L1 cache associativities</td>
<td>2-way</td>
<td>4-way</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>none</td>
<td>256 K</td>
</tr>
<tr>
<td>DRAM access latency</td>
<td>54 cycles</td>
<td>86 cycles</td>
</tr>
</tbody>
</table>

\(^{a}\)Note that despite the area penalty for an ASIC implementation, Raw is almost certainly a larger design than the P3. This evaluation does not aim to make a cost-normalized comparison, but rather seeks to demonstrate the scalability of the tiled multicore approach for future microprocessors.

**System Models:** After the selection of a reference processor comes the selection of an enclosing system. A pair of 600 MHz Dell Precision 410 machines were used to run the reference benchmarks. These machines were outfitted with identical 100 MHz 2-2-2 PC100 256 MB DRAMs, and several microbenchmarks were used to verify that the memory system timings matched.

To provide a fair comparison between the Raw and Dell systems, the Raw simulator extension language was used to implement a cycle-matched PC100 DRAM model and a chipset. This model has the same wall-clock latency and bandwidth as the Dell 410. However, since Raw runs at a slower frequency than the P3 (425 MHz vs. 600 MHz), the latency, measured in cycles, is less. The term **RawPC** is used to
describe a simulation which uses eight PC100 DRAMs, occupying four ports on the left hand side of the chip, and four on the right hand side.

Because Raw is designed for streaming applications, it is necessary to measure applications that use the full pin bandwidth of the chip. In this case, a simulation of CL2 PC 3500 DDR DRAM, which provides enough bandwidth to saturate both directions of a Raw port, was used. This is achieved by attaching 16 PC 3500 DRAMs to all 16 logical ports on the chip, in conjunction with a memory controller, implemented in the chipset, that supports a number of stream requests. A Raw tile can send a message over the general dynamic network to the chipset to initiate large bulk transfers from the DRAMs into and out of the static network. Simple interleaving and striding is supported, subject to the underlying access and timing constraints of the DRAM. This configuration is called RawStreams.

The placement of a DRAM on a Raw port does not preclude the use of other devices on that port—the chipsets have a simple demultiplexing mechanism that allows multiple devices to connect to a single port.

Other Normalizations: Except where otherwise noted, gcc 3.3 –O3 was used to compile C and Fortran code for both Raw\(^2\) and the P3\(^3\). For programs that do C or Fortran stdio calls, newlib 1.9.0 was used for both Raw and the P3. Finally, to eliminate the impact of disparate file and operating systems, the results of I/O system calls for the Spec benchmarks were captured and embedded into the binaries as static data using [41].

One final normalization was performed to enable comparisons with the P3. The cycle-accurate simulator was augmented to employ conventional two-way set-associative hardware instruction caching. These instruction caches are modeled cycle-by-cycle in the same manner as the rest of the hardware. Like the data caches, they service misses over the memory dynamic network. Resource contention between the caches is modeled accordingly.

1.4.2 ILP Computation

This section examines how well Raw is able to support conventional sequential applications. Typically, the only form of parallelism available in these applications is instruction-level parallelism (ILP). For this evaluation, a range of benchmarks that encompasses a wide spectrum of program types and degrees of ILP is used.

Much like a VLIW architecture, Raw is designed to rely on the compiler to find and exploit ILP. However, unlike a VLIW, it does so by distributing instructions across multiple processor cores with independent program counters. This process of distributing ILP across multiple cores is called distributed ILP or DILP. We have developed Rawcc [8, 25, 26] to explore DILP compilation issues. Rawcc takes sequential C or Fortran programs and orchestrates them across the Raw tiles

\(^2\)The Raw gcc backend targets a single tile’s compute and network resources.
\(^3\)For the P3, the –march=pentium3 and –mfpmath=sse flags were added.