The Dissertation Committee for Changkyu Kim

certifies that this is the approved version of the following dissertation:

A Technology-Scalable Composable Architecture

Committee:

Douglas C. Burger, Supervisor

James C. Browne

Stephen W. Keckler

Kathryn S. McKinley

Charles R. Moore

A Technology-Scalable Composable Architecture

by

Changkyu Kim, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

To my loving wife, Eunyoung Park

And my mother, Inja Yum

Acknowledgments

First and foremost, I am forever grateful to my advisor, Doug Burger for his constant support, advice and patience. I thank Doug Burger for being not only a great research advisor, but an excellent mentor. He has believed in me from day one, and always encouraged me to aim at higher goals than I could think of. It has been always great pleasure working with him and I am truly fortunate and at the same time proud to have him as my advisor. I hope I can make him proud, too, throughout my future career.

I am deeply indebted to my co-advisor Steve Keckler for his technical expertise and guidance. His relentless passion and research vision has greatly influenced me. I cannot thank him enough for his valuable advice and insights that inspired this dissertation work.

I am very thankful to the other members of my Ph.D. committee, Kathryn McKinley, Chuck Moore and James Browne for their helpful feedback on my thesis research. Special thanks go to Kathryn McKinley for her thoughtful advice and technical comments on this dissertation work.

The work described in this dissertation would not have been possible without the help of members in the CART research group. I especially thank Karu Sankaralingam for having numerous technical and non-technical conversation and keeping me healthy by exercising together. I would like to give special thanks to Jaehyuck Huh who helped me to

settle in and has been good company.

Many thanks are extended to Heather Hanson and Sadia Sharif for offering help on proofreading dissertation drafts; Ramadass Nagarajan for answering my tedious questions on simulators and benchmarks patiently; Premkishore Shivakumar and Simha Sethumadhavan for sharing joys and difficulties together ever since we all started doctoral study at the same time; Madhu Sibi Govindan for bringing humor every day and being a wonderful cubicle mate; Paul Gratz for being an great project partner when we worked together on the TRIPS project weekendless; Nitya Ranganathan, Haiming Liu, and Divya Gulati whom I collaborated with in the TFlex study; Bert Maher, Katie Coons and Mark Gebhart for helping with compiler infrastructure and performance evaluation.

I also thank Pradeep Dubey, Yen-Kuang Chen, and Chris Hughes for giving me a great internship opportunity at Intel. I would also like to acknowledge that this work is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF CAREER grants CCR-9985109 and CCR-9984336, NSF grant EIA-0303609, two IBM University Partnership awards, and a grant from the Intel Research Council.

Last, but not the least, I would like to thank my mother, Inja Yum for her unconditional sacrifice and love throughout my life. My mom has truly been a blessing and I thank her for her many prayers and support throughout my entire life. I would like to deeply thank my brother, Hyungkyu Kim, who has been my best friend and great supporter now, for keeping good company with my mom since I came here in US to start doctoral study. I thank my loving wife and the best friend, Eunyoung Park, for her constant love and emotional support throughout. She brought so much happiness and warmth in my life. I look forward to our tomorrow and the next day, as we grow old together. Without my mom, my brother, and my wife, I would not be what I am now. I dedicate this dissertation to them.

CHANGKYU KIM

The University of Texas at Austin

August 2007

A Technology-Scalable Composable Architecture

Publication No. _____

Changkyu Kim, Ph.D.

The University of Texas at Austin, 2007

Supervisor: Douglas C. Burger

Clock rate scaling can no longer sustain computer system performance scaling due to power and thermal constraints and diminishing performance returns of deep pipelining. Future performance improvements must therefore come from mining concurrency from applications. However, increasing global on-chip wire delays will limit the amount of state available in a single cycle, thereby hampering the ability to mine concurrency with conventional approaches.

To address these technology challenges, the processor industry has migrated to chip multiprocessors (CMPs). The disadvantage of conventional CMP architectures, however, is their relative inflexibility to meet the wide range of application demands and operating targets that now exist. The granularity (e.g., issue width), the number of processors in a chip and memory hierarchies are fixed at design time based on the target workload mix, which result in suboptimal operation as the workload mix and operating targets change over time.

In this dissertation, we explore the concept of *composability* to address both the

increasing wire delay problem and the inflexibility of conventional CMP architectures. The basic concept of *composability* is the ability to dynamically adapt to diverse applications and operating targets, both in terms of granularity and functionality, by aggregating fine-grained processing units or memory units.

First, we propose a composable on-chip memory substrate, called Non-Uniform Access Cache Architecture (NUCA) to address increasing on-chip wire delay for future large caches. The NUCA substrate breaks large on-chip memories into many fine-grained memory banks that are independently accessible, with a switched network embedded in the cache. Lines can be mapped into this array of memory banks with fixed mappings or dynamic mappings, where cache lines can move around within the cache to further reduce the average cache hit latency.

Second, we evaluate a range of strategies to build a composable processor. Composable processors provide flexibility of adapting the granularity of processors to various application demands and operating targets, and thus choose the hardware configurations best suited to any given point. A composable processor consists of a large number of low-power, fine-grained processor cores that can be aggregated dynamically to form more powerful logical processors. We present architectural innovations to support composability in a power- and area-efficient manner.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past two decades, the continuing scaling of CMOS devices and aggressive pipelining achieved a 40% per year increase in clock speeds: from 33MHz in 1990 to over 3GHz in 2004, and contributed the bulk of the performance growth during the same period. However, recent trends show that doubling of clock frequencies every two years has come to an end as power dissipation and thermal issues become first-order design constraints [46, 82, 106], and as pipeline depths have reached their practical limits [45, 51]. This technology trend heralds the end of the frequency scaling era. Intel canceled its high frequency Pentium 4 successors [125], and major processor companies have announced multicore architectures for future microprocessor designs, which are further evidence of the shift into the concurrency era. Therefore, most performance improvements in future systems must come from power-efficient exploitation of concurrency.

Another technology trend is that the delay of on-chip global wires grows relative to the delay of gates [50, 77]. The increasing wire delay has already affected traditional microarchitectures. For example, the Intel Pentium 4 assigns two separate pipeline

stages (called “drive” stages) among the total 20 stages for routing information around a chip [49]. In addition, the single uniform access latency seen in traditional large on-chip caches has changed into different latencies depending on the physical location of data within the cache [88, 120]. While the recent trend of decelerating frequency growth may lessen the effect of wire delays, increasing resistive delay through global on-chip wires will allow only a small fraction of a chip to be reachable within a single cycle [2], and thus limit the ability to mine concurrency with conventional approaches. Eventually, increasing global on-chip wire delays will force architectures to become communication driven and inherently distributed [89]. Future architectures must therefore address wire delays explicitly to achieve high performance.

1.1 Microarchitecture Configuration for Optimal Points

Good microarchitecture configurations are affected by the following two variables, workload characteristics and operating targets (metrics).

- Workload Diversity: Over the last decades, application domains have become increasingly diversified, now including desktop, network, server, scientific, graphics and digital signal processing. In each domain, applications have different granularities of concurrency and place different demands on underlying hardware. Moreover, many future applications such as video databases are expected to have heterogeneous computational requirements [27]. In addition to diverse granularities of concurrency, applications have diverse memory requirements. First, applications from different domains have different memory access patterns [10]. Traditional desktop and enterprise applications tend to have more irregular access patterns, while scientific and

graphics applications typically have regular and streaming access patterns [72]. Second, the required size of working sets vary across different applications or different execution phases within the same application [1, 4, 26, 91, 96].

- Operating Targets: A single application can benefit from multiple distinct hardware configurations depending on operating targets (or metrics) [41]. The operating targets depend on what we wish to optimize, including the shortest execution time of a single-threaded application, maximum throughput, power, energy, or the energy-delay product. For example, to maximize performance of a single-threaded application, the whole system needs as many hardware resources as possible to be assigned for the application, thus leading to a processor design with few, but large, aggressive cores. To maximize throughput under abundant threads, the system needs to maximize performance per unit area and should favor many small cores. A similar argument can be applied to maximizing power efficiency [81]. DVFS (dynamic voltage-frequency scaling) can be used to address various power-performance needs without changing hardware configurations [73]. However, powering down or reducing the voltage/frequency of unused structures cannot reduce the power consumption as much as designing a smaller core to begin with [6], which motivates determining the right hardware configurations depending on the target performance-power-throughput profiles.

Despite diverse workload characteristics and operating targets, conventional processors and cache architectures have a rigid granularity, meaning that designers must fix the granularity of processors and balance the capacity and access time of each cache hierarchy based on the intended workload mix. This fixed granularity of processors and cache hierarchy will typically result in either performance or power loss (or both) outside the target

application mix and intended operating settings.

To handle these two types of diversity, future microarchitectures should change configurations to extract different levels of concurrency efficiently and provide optimized working points at different operating targets. The changing of hardware configurations includes both allocating different amounts of hardware resources (e.g, issue with, issue window size and cache capacity, etc.) and providing different types of hardware organizations (e.g, cache memory or scratchpad memory).

1.2 Other Approaches

The recent reduction in frequency scaling rates implies that most performance improvements in the future will come from exploiting more concurrency. Concurrency can be exploited by many levels of modern systems: by hardware (ILP/superscalar processors [109]), by support in the ISA and compiler (VLIW architectures [87]), or by the compiler [71] or programmer [104] in parallel systems. Since superscalar and VLIW processors' widths have not scaled recently due to growing wire delays, increasing design complexity, and power constraints, industry has migrated toward chip multiprocessors (CMPs) composed of moderately complex cores and is hoping that software threads will provide the needed concurrency. However, such a solution has the several limitations.

First, while conventional chip multiprocessors offer a power-efficient way to mine concurrency from parallel workloads, the serial execution portion of these parallel workloads or the single-threaded workloads tend to be limited by the performance of single core in CMPs (that can sustain modest ILP). Unless a programmer or a compiler parallelizes the code (an approach that has produced only limited success for past decades), Amdahl's law ultimately hampers the overall system performance growth. Second, current CMP de-

signs have fixed granularity, meaning that the size of a processor core and the number of processor cores in a chip are fixed at design time. Any such fixed design point will result in suboptimal operation in terms of either performance or power (or both) across a diverse workload mix due to the varied granularity of types of concurrency.

One alternative design to alleviate inefficiency caused by these diversities is integrating multiple heterogeneous processor cores that are tuned to specific applications in a single die. The "Single-ISA Heterogeneous Multi-Core Architecture" work by Kumar et al. [66, 68] or the "Asymmetric Chip Multiprocessors" work by Balakrishnan et al. [9] is one approach to address the diversity problem. Their approach is to build a chip multiprocessor out of cores of various sizes and performance profiles. While a large processor core speeds up a sequential region of code or application with fewer threads, many small processor cores collectively run parallel software. The design complexity also can be reduced by reusing the off-the-shelf processor cores from previous generations. However, the particular processor core composition is still fixed at design time, which may cause inefficiency outside the target workload mix. Another challenge in terms of manufacturability is the difficulty of silicon integration of heterogeneous cores. When processor cores, with different performance profiles, from previous generations are integrated, they require different manufacturing processes [6, 128].

Another approach to integrated heterogeneity is to build a heterogeneous chip that contains multiple different cores, each designed to run a distinct class of workloads effectively. The Tarantula processor [29] and the IBM Cell [59] are good examples of this approach. While such specialization provides application-specific processor efficiency, the increased design complexity caused by the poor design reuse is one of the main drawbacks. More importantly, programming on application-specific heterogeneous cores poses a sig-

nificant - and in some cases intractable - programming challenge [6].

1.3 Principles of Composable Architecture

To address both current technology challenges and diverse application demands, we evaluate a range of techniques to build a technology-scalable composable architecture. First, we define *composability* as *the ability to adapting underlying hardware resources dynamically to different applications or operating targets, by aggregating fine-grained processing units or memory units*. The main principles of composable architectures include the following, which are developed in the remainder of this dissertation.

- Composable architectures are built on a distributed substrate consisting of multiple fine-grained processing and memory units. The fine-grained units are inherently more power-efficient and achieve technology scalability with respect to future global wire delay increases.

- Composable architectures provide the ability (1) to aggregate fine-grained units to compose into larger logical units and (2) to match each application to the composed logical unit best suited to meet its performance, power, and throughput demands.

- The number of fine-grained units combined to execute each application can be dynamically changed transparently to the running application.

- Composable architectures need to provide an ISA and microarchitectural support that combines distributed fine-grained units in a power- and area-efficient manner. The area and complexity to support composability in a distributed substrate should be minimized.

1.4 Thesis Statement

This dissertation introduces the concept of *composability*: The aggregation of fine-grained units to adapt to diverse application demands and different operating targets (metrics). Compared to monolithic, coarse-grained units, the fine-grained units are inherently more power-efficient and provide further opportunities to optimize power consumptions with finer-granularity. In addition, the fine-grained units are more tolerant to future wire-delay dominated technologies. This dissertation presents architectural innovations to support composability that provides the flexibility to allocate resources dynamically to different types of concurrency and various working set sizes. Specifically, this dissertation first proposes a novel level-2 cache design to address the increasing global on-chip wire delay problem for future large on-chip caches. Second, this dissertation describes ISA and microarchitectural support for run-time configuration of fine-grained CMP processors, allowing flexibility in aggregating cores together to form larger logical processors.

1.5 Dissertation Contributions

This dissertation evaluates composable architectures that have two main components: Composable memory systems and composable processors.

1.5.1 Composable Memory Systems

- Future increases in on-chip global wire delays will make the uniform access time of traditional large on-chip caches untenable. Data residing in the part of a large cache close to the processor can be accessed much faster than data that reside farther from the processor. In this dissertation, we explore cache designs that can exploit the non-

uniformity of cache access times among banks of a single cache and evaluate two different cache substrates depending on types of interconnection network between multiple cache banks. We call these new cache substrates for future wire-delay dominated technologies, Non-Uniform Cache Architecture (NUCA).

- The non-uniform access latency in future large caches can be further exploited by dynamically migrating important data so that the working sets are clustered near the processor. By permitting data to be mapped to one of many banks within the cache, and to migrate among them, a cache can be automatically managed in such a way that most requests are serviced by the fastest bank (the closest bank to the processor). This dynamic migration capability allows caches to adapt to applications with various working set sizes, thereby eliminating the trade-off between larger, slower caches for applications with large working sets, and smaller, faster caches for applications that are less memory intensive.

- Applications from various domains have different memory access patterns, and thus require various memory organizations. For example, while applications that have irregular access patterns will get more benefits from cache memories, streaming applications from the scientific and graphics domains can exploit scratchpad memories. The composable memory system that we evaluate in this dissertation provides a flexible substrate that can be reconfigured into various memory organizations because it consists of multiple fine-grained memory banks connected by a on-chip switched network. Each memory bank can be configured differently (either cache memory or scratchpad memory) and be aggregated to form various memory organizations depending on the running applications. As a proof of concept, we built a composable secondary memory system in the TRIPS prototype [99]. The TRIPS secondary mem-

ory system supports a wide range of memory organizations from a 1MB L2 cache, to a 1MB scratchpad memory, to any combination in between at the granularity of 64KB increments.

- The trend of integrating many processor cores in a chip multiprocessor (CMP) provides a new challenge in designing the on-chip memory system. Even though L1 caches in CMPs are likely to remain private and be tightly integrated to the processor cores, the question of how to manage the L2 caches will be key to building a scalable CMP. The L2 caches may be shared by all processors or may be separated into private per-processor partitions. While the private L2 design offers faster access time than the shared L2 design, the shared L2 design can reduce the number of critical off-chip misses with a larger effective cache size. In this dissertation, we address the slow hit time in the shared L2 design with the dynamic working set clustering capability that we explored in the uniprocessor context, and thus achieve both the benefits of the private L2 design and the shared L2 design.

- Jaehyuk Huh and I jointly worked to extend the NUCA L2 design to CMP L2 caches. Jaehyuk Huh led the project and developed the CMP simulators focusing on the effect of various sharing degrees on cache performance. I explored the effect of dynamic data migration in CMP L2 caches in terms of both performance and energy.

1.5.2 Composable Processors

- The processor industry has migrated toward CMPs because of thermal and power constraints, but the current CMP designs have significant drawbacks. Current CMP designs have a fixed granularity, meaning that the number and capabilities of the processors are rigid. This fixed granularity will result in suboptimal operation outside

the intended target domain, and thus either performance or power efficiency (or both) will suffer. In this dissertation, we explore a composable CMP called "Composable Lightweight Processors" (or CLPs) that provides flexibility of adapting the granularity of processors to various application characteristics and operating targets. A CLP consists of a large number of low-power, fine-grained processor cores that can be aggregated dynamically to form more powerful, single-threaded logical processors.

- While composability can also be provided using traditional ISAs [15], we examine CLPs in the context of an Explicit Data Graph Execution (EDGE) ISA [54] that provides the following salient features for composability. First, when a single-threaded application runs on multiple distributed cores, traditional architectures will require careful coordination among cores to maintain the sequential semantics of the instruction stream, especially in the in-order stages of pipelines, such as fetch and commit. This coordination overhead can be significantly reduced if the unit of coordination is done at a granularity larger than individual instructions. EDGE ISAs allow the hardware to fetch, execute, and commit blocks of instructions, rather than individual instructions, in an atomic fashion. Second, EDGE ISAs support dataflow execution within a block, by specifying the inter-instruction data dependence relationship explicitly. Since the dataflow graph is explicitly encoded in the instruction stream, it is simple to shrink or expand the graph on a smaller or greater number of execution resources as desired with little additional hardware.

- The microarchitectural structures in a composable processor require capabilities different from those available in some of the microarchitectural structures of traditional superscalar processors. These capabilities must permit composable microarchitectural structures to be incrementally added or removed as the number of participating

cores increases or decreases. Ideally, the area and complexity to support composability should be kept low so as not to increase the power or area overhead needed to support composability. In particular, the hardware resources should not be oversized or undersized to suit either a large processor configuration or a small configuration. Additionally, centralized structures that will limit the scalability of the microarchitecture must be avoided.

To provide this capability, we identify and repeatedly apply two principles. First, the microarchitectural structures are partitioned by address wherever possible. Since addresses of both instructions and data tend to be equally distributed, address partitioning ensures (probabilistically, at least) that the useful capacity increases/decreases monotonically. Second, we avoid physically centralized microarchitectural structures completely. Decentralization allows the structure sizes to be grown without the undue complexity traditionally associated with large centralized structures. In this dissertation, we evaluate the overheads to support composability in a distributed substrate and show that the proposed CLP microarchitecture using the EDGE ISA keeps these overhead sufficiently low.

- This dissertation summarizes some of microarchitectural mechanisms that are the subject of several dissertations including the distributed branch predictor by Ranganathan [95], the distributed instruction fetch by Liu [74], and the distributed memory disambiguation by Sethumadhavan [103], and are covered in detail in their respective dissertations.

- This dissertation demonstrates that the best processor configuration is quite different depending on application characteristics and operating targets — performance, area

efficiency, power efficiency. Our proposed CLP architecture provides the ability to shift to different processor configuration when the need arises.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 evaluates two different cache substrates for composable memory systems depending on types of interconnection networks that connect multiple cache banks — one with private per-bank channels, the other using an on-chip switched network. Then, we describe an implementation of a composable secondary memory system in the TRIPS prototype. The TRIPS secondary memory system exploits the configurable nature of switched networks to allow various memory organizations on the same cache substrate.

Chapter 3 describes the dynamic mapping mechanisms supported in a composable cache substrate and presents the performance effect in the context of a uniprocessor design. Then, we extend this concept of dynamic mapping to L2 caches in chip multiprocessors and investigate the effect of dynamic migration capabilities within the same cache hierarchy on both the average hit latency and the energy consumed by the L2 cache subsystem.

Chapter 4 describes strategies for composing processors that aggregates lightweight EDGE cores to form larger, more powerful logical single-threaded processors when the need arises. We show that the composable lightweight processors provide the ability to expand or shrink the granularity of processor and adapt to different metrics such as performance, area efficiency and power efficiency. Chapter 5 presents a summary of the overall contributions of this dissertation and future work.

Chapter 2

Related Work

This chapter discusses and differentiates prior work most closely related to the focus of this dissertation. We present the prior work as it relates to the two main components of this dissertation: (1) composable processors (2) composable on-chip memory systems.

2.1 Composable Processors

The ability to adapt multiprocessor hardware to fit needs of the available software is clearly desirable, both in terms of overall performance and power efficiency [5, 53]. The amount of prior research that address this problem has been considerable, and we categorize prior research into four broad categories. In the first, researchers attempt to provide higher single-thread performance from a collection of distributed units. In the second, researchers design large cores and provide the capability to resize or share subcomponents of the processor. In the third, researchers explicitly implement multiple distinct granularities to allow software to choose the appropriate hardware. In the fourth, researchers build a single programmable substrate that can be reconfigured to match the different granularity of concurrency.

2.1.1 Composing Processors from Smaller Cores

Many research efforts have attempted to synthesize a more powerful core out of smaller or clustered components.

The recent Core Fusion [54] work is most similar to the CLP approach. Core Fusion consists of multiple, 2-wide, relatively simple out-of-order cores connected by a bus. Like CLPs, Core Fusion allows multiple dynamically allocated processors to share a single contiguous instruction window. The goal is to accommodate software diversity and support incremental parallelization by dynamically providing the optimal configuration for sequential and parallel regions of programs. The advantage of Core Fusion is that it exploits conventional RISC or CISC ISAs and maintains software compatibility. In a Core Fusion implementation, several structures must be physically shared, limiting the range of composition up to four cores (8-wide issue).

First, while each core accesses its own I-cache to fetch instructions and facilitates collective fetch, the centralized fetch management unit (FMU) handles the resolution of control-flow changes among any participating cores. Every time a core predicts a taken branch or detects a branch misprediction, it sends the new target PC to the FMU. The FMU collects the target PC information and broadcasts the redirected control-flow to all participating cores. Second, the centralized steering management unit (SMU) takes care of renaming and steering to track dependence information across different cores and keep the dependent instructions close. After pre-decode, each core sends two instructions to the SMU, which must support the renaming of up to eight instructions each cycle for a four-core fused operation. To steer eight instructions every cycle, the SMU requires an eight-stage rename pipeline and a steering table that has sixteen read and sixteen write ports. These physically shared, multi-ported, centralized structures limit the maximum supported

composition ranges. The TFlex CLP shares no resources physically, so it can scale up to 64-wide issue, but relies on a non-standard EDGE ISA to achieve full composability.

Clustered superscalar processors [16] and the compiler-supported multi-cluster design [30] both aim to improve the scalability of a large, out-of-order superscalar processor by using multiple, clustered execution resources. While this approach decreases the complexity of each cluster, it shares the disadvantage that adaptive processing has of with respect to its inability to trade off multiple threads for core granularity.

Most other prior work that attempts to synthesize a large logical processor from smaller processing elements uses independent sequencers with a non-contiguous instruction window. An early example is the Multiscalar architecture [112]. Multiscalar processors used speculation to fill up independent processing elements (called *stages*), with each of the speculative stages starting from a predicted, control-independent point in the program. The Multiscalar design used a shared resource (the ARB) for memory disambiguation and did not permit the stages to run distinct software threads independently. The subsequent speculative threads work [43, 65] adapted the Multiscalar execution model to a CMP substrate that could execute separate threads on the individual processors when not in speculative threads mode. The CLP approach that we explore in this chapter differs from such architectures in that CLPs employ a single logical point of control, i.e., a contiguous instruction window, across the multiple processing elements, which simplifies dependence tracking.

Other composable approaches have provided statically exposed architectures that can be partitioned. The best example is the RAW architecture [119], an important and early tiled architecture. The RAW compiler can target any number of single-issue RAW tiles, forming a single static schedule across them. Each tile still has its own instruction sequencer, although they are highly synchronized with one another. Multiple tasks can

be run across a set of tiles provided that each task was compiled for the number of tiles to which it was allocated. While RAW requires recompiling applications for changing configurations, CLPs achieve this configurability transparent to the software.

2.1.2 Partitioning large cores

The most popular approach for partitioning large cores to date has been Simultaneous Multithreading [121], in which multiple threads share a single large, out-of-order core. The operating system achieves adaptive granularity by adjusting the number of threads that are mapped to one processor. The advantages of SMT are extremely low overheads for providing the adaptive granularity. A disadvantage is the limited range of granularity since processors are typically restricted to be four-wide, and threads sharing the same core may cause significant interference. In addition, resources in an SMT processor may be underutilized leading to unnecessary power consumption overhead when executing a single-threaded application that can achieve competitive performance with a less complex processor core.

What Albonesi has termed "adaptive processing" [3] involves dynamically resizing large structures in an out-of-order core, powering fractions of them down based on expected requirements, thus balancing power consumption with performance by efficiently mapping threads to right-sized hardware structures. Researchers have proposed adjusting cache size via ways [4], issue window size [34], the issue window coupled with the load/store queue and register file [90], and issue width, along with the requisite functional units [7]. While adaptive processing permits improved energy efficiency by adjusting the core's resources to the needs of the running application, it does not permit a fine-grained tradeoff between core granularity and number of threads. While combining adaptive processing with SMT might achieve that goal, the complexity and overheads on a large-centralized core would be

significant.

Finally, conjoined-core chip multiprocessing [67] aims to provide some shared resources, with other explicitly partitioned resources, effectively creating a hybrid between SMT and CMP approaches. Conjoined-core CMP is built on a CMP substrate and allows resource sharing between adjacent cores to reduce die area with minimal performance loss and thus improves the overall computational efficiency. The authors investigate the possible sharing of floating-point units, crossbar ports, first-level instruction caches, and first-level data caches. To minimize area overheads and design complexity, conjoined-core CMP only allows resource sharing between adjacent pairs of processors. Therefore, similar to SMT approaches, the degree of granularity configuration between single threads versus multiple threads is more limited than the CLP approach explored in this chapter.

2.1.3 Multiple Granularities

Some proposals aim to match an application's granularity needs by providing the hardware that best suits the application. The "Single-ISA Heterogeneous Multi-Core Architecture" work by Kumar et al. [66, 68] or the "Asymmetric Chip Multiprocessors" work by Balakrishnan et al. [9] is to build a chip multiprocessor out of cores of various sizes with different performance profiles. Single-ISA, heterogeneous multi-core architectures [66] reuse a discrete number of processor cores that were implemented across multiple previous generations with each having different issue width, cache sizes, and characteristics (e.g, in-order vs. out-of-order). On the other hand, asymmetric chip multiprocessors consists of processor cores with the same size, but introduces heterogeneity across different cores by changing the duty cycle of the processor for thermal management. Their goal is to integrate the various granularities of processors to better exploit both variations in thread-level parallelism as

well as inter- and intra-thread diversity to increase both performance and energy efficiency. With this approach, a large (or faster) processor core speeds up a sequential region of code or application with fewer threads and many small (or slower) processor cores collectively run parallel software.

Both these approaches increase design complexity and limits the number of granularity options. Therefore, for example, while a large, complex core can increase performance on sequential code, it may do so at the expense of performance of parallel applications. However, this approach does not suffer from the overhead of making the processors variable-grain or composable.

2.1.4 Reconfigurability

Researchers have also explored a single programmable substrate that can be reconfigured to match the different granularities of concurrency.

FPGAs provide the finest granularity for reconfiguration. FPGAs consist of an array of gates or programmable lookup tables interconnected through a configurable network. While using FPGAs can offer high performance with fine-grained data parallelism per application, achieving good performance on general-purpose and serial applications has not been shown to be feasible.

Coarse-grained reconfiguration architectures stress the use of coarse grain reconfigurable arrays to address the huge routing area overhead and poor routability of ultra fine-grained FPGAs [44]. Fisher et al. proposed custom-fit processors to choose the right grain size for specific applications at design time [32]. Similarly, Tensilica Xtensa customizes processor cores at design time for a given application [37]. Xtensa is built on a synthesizable processor that can customize I-, D- cache sizes, number of registers, data RAM

size, and external bus width at design time. In addition, Xtensa provides the capability of extending instruction sets to allow application-specific functionality. These coarse-grained reconfiguration approaches clearly increase application-specific efficiency at the expensive of run-time flexibility.

The following architectures were proposed to exploit the different granularity of concurrency on a single substrate. Compared to FPGAs and coarse-grained reconfigurable architectures, these novel architectures can support general-purpose sequential programs. Browne et al. developed the Texas Reconfigurable Array Computer (TRAC) that supports both SIMD and MIMD processing by reprogramming interconnections between individual processing elements and memory elements [57, 100]. The Stanford Smart Memories architecture can reconfigure processors and memories in addition to interconnections and match various application characteristics [76]. The Stanford Smart Memories support coarse-grained reconfiguration capabilities that allow diverse computing models, like speculative multithreading and streaming architectures. Sankaralingam defined the concept of *architectural polymorphism* and explored a set of mechanisms that configure coarse-grained microarchitecture blocks to support different granularity of parallelism in the context of the TRIPS processor [97]. He formally defined architectural polymorphism as: "the ability to modify the functionality of coarse-grained microarchitecture blocks at runtime, by changing control logic but leaving datapath and storage elements largely unmodified, to build a programmable architecture that can be specialized on an application-by-application basis."

2.2 Composable On-chip Memory System

There is much prior research in addressing the increasing global on-chip wire delay problem in future large caches. We first discuss related work in the context of uniprocessor systems

and then extend the discussion in the context of chip multiprocessor systems.

2.2.1 Uniprocessor Level-2 Caches

Prior work has evaluated large cache designs, but not for specifically wire-dominated technologies; Kessler examined designs for multi-megabyte caches built with discrete components [60]. Hallnor and Reinhardt [42] studied a fully associative software-managed design, called "Indirect Index Cache" (or IIC), for large on-chip L2 caches. The IIC does not co-locate a tag with a specific data block; instead, each tag contains a pointer to locate the corresponding data block. This indirection allows large level-2 caches to be implemented with a fully-associative cache amenable to software management. However, the IIC did not consider non-uniform access latencies of a large cache.

Other work has examined using associativity to balance power and performance. Albonesi examines turning off "ways" of each set to save power when cache demand is low [4]. He proposes the cache structure that provides the ability to dynamically enable a subset of data ways on demand, thus reducing the switching activity of the cache. Powell et al. use way-prediction to predict the matching way number, instead of waiting on the tag array to provide the way number by sequential tag access. Since low energy consumption can also be achieved when prediction is correct, they evaluate the balance between incremental searches of the sets to balance power and performance [91].

Other researchers have examined using multiple banks for high bandwidth, as we do to reduce contention. Sohi and Franklin [113] proposed interleaving banks to create ports, and also examined the need for L2 cache ports on less powerful processors than today's. Wilson and Olukotun [123] performed an exhaustive study of the trade-offs involved with port and bank replication and line buffers for level-one caches. This dissertation aims to

flatten deepening hierarchies; a goal that should be compared with Przybylski's dissertation, in which he exhaustively searched the space of multi-level caches to find a performance-optimal point [92].

Non-uniform accesses are appearing in high performance cache designs [88]. The following two studies investigated ways to handle increasing the global on-chip wire delay problem in large L2 caches. Beckmann and Wood proposed the Transmission Line Cache (TLC) to replace long wires in large uniprocessor caches with LC transmission lines for reducing wire delay [11]. Chishti et al. investigate the dynamic data migration to exploit non-uniform access latencies in a large cache and extend our study on non-uniform access cache architectures. The main difference is that they proposed decoupling data placement from tag placement to contain more data from "hot" sets which have the same index [19]. They used the coarser grained distance group to reduce the energy consumption caused by migrating data.

2.2.2 Chip Multiprocessor Level-2 Caches

Shared caches have been studied in the context of chip multiprocessors and multithreaded processors. Nayfeh et al. investigated shared caches for primary and secondary caches on a multi-chip module substrate with four CPUs [85]. They examined how the memory sharing patterns of different applications affect the best cache hierarchy. Subsequent work from the same authors examined the trade-offs of shared-cache clustering in multi-chip multiprocessors [86]. With eight CPUs, they observed that the coherence bus becomes the performance bottleneck for private L2 caches, suggesting the utility of shared caches to reduce bus traffic.

Recent studies considered wire latency as a primary design factor in CMP caches.

Beckmann and Wood compared three latency reduction techniques including dynamic block migration, L1/L2 prefetching, and faster on-chip transmission lines with an 8-CPU shared cache [12]. They conclude that data migration is less effective for CMPs because each sharer pulls the data towards it, leaving the block in the middle, far away from all sharers. Chishti et al. study optimizations with NuRAPID cache designs to reduce unnecessary replication and communication overheads [20]. Zhang et al. [127] proposed the victim replication cache design which selectively keeps copies of primary cache victims in each local L2 slice. Both NuRAPID and victim replication designs attempt to reduce the latency further by allowing replication, while our study relies on migration and maintains a single copy of data within the L2 cache to save on-chip capacity. The NuRAPID and victim replication designs have different replication policies; NuRAPID replicates data on access and victim replication replicates data on eviction. While the above three cache designs are based on a shared L2 cache, Cooperative Caching [17] uses private caches as the baseline design and adopts the benefits of a shared cache by using cache-to-cache transfers and modifying cache replacement policies. Lastly, Speight et al. studied how CMP L2 caches interact with off-chip L3 caches and how on-chip L2 caches temporarily absorb modified replacement blocks from other caches [114].

Chapter 3

Composable On-Chip Memory Systems

Historically, the capacity of on-chip level-two (L2) caches has been limited by the available number of transistors in a chip. The persistent growth in on-chip transistor counts following Moore's law increased L2 cache capacity over time. The Alpha 21164, introduced in 1994, had 96KB on-chip L2 cache [28], while today's high performance processors incorporate larger L2 caches (or even L3 caches) on the processor die. The HP PA-8700 contains 2.25MB of unified on-chip cache [48], and the Intel Montecito Itanium contains 6MB of on-chip L3 cache [79]. The sizes of on-chip L2 and L3 cache memories are expected to continue increasing as the bandwidth demands on the package grow, and as smaller technologies permit more bits per mm^2 [53].

Current multi-level cache hierarchies are organized into a few discrete levels. Typically, each level obeys inclusion, replicating the contents of the smaller level above it, and reducing accesses to the lower levels of the cache hierarchy. When choosing the size of each

level, designers must balance access time and capacity, while staying within area and cost budgets. In future technologies, large on-chip caches with a single, discrete hit latency will be undesirable, due to increasing global wire delays across the chip [2, 77]. Data residing in the part of a large cache close to the processor could be accessed much faster than data that reside physically farther from the processor.

In this chapter, we explore the design space for composable on-chip memory substrates in future wire-delay dominated technologies. We first show that traditional cache designs, in which a centralized decoder drives physically partitioned sub-banks, will be ineffective in future technologies, as data in those designs can be accessed only as fast as the slowest sub-bank. We evaluate multiple composable on-chip memory substrates in which large on-chip memories are broken into many fine-grained memory banks that can be accessed at different latencies.

Figure 3.1 shows the types of organizations that we explore in this chapter, listing the number of banks and the average access times, assuming 16MB caches modeled with a 45nm technology. The numbers superimposed on the cache banks show the latency of a single contentionless request, derived from a modified version of the Cacti [105] cache modeling tool. The average loaded access times shown below are derived from performance simulations that use the unloaded latency as the access time but which include port and channel contention.

We call a traditional cache a Uniform Cache Architecture (UCA), shown in Figure 3.1a. Even with aggressive sub-banking, our models indicate that this cache would perform poorly due to internal wire delays and restricted numbers of ports.

Figure 3.1b shows a traditional multi-level cache (L2 and L3), called ML-UCA. Both levels are aggressively banked for supporting multiple parallel accesses, although the

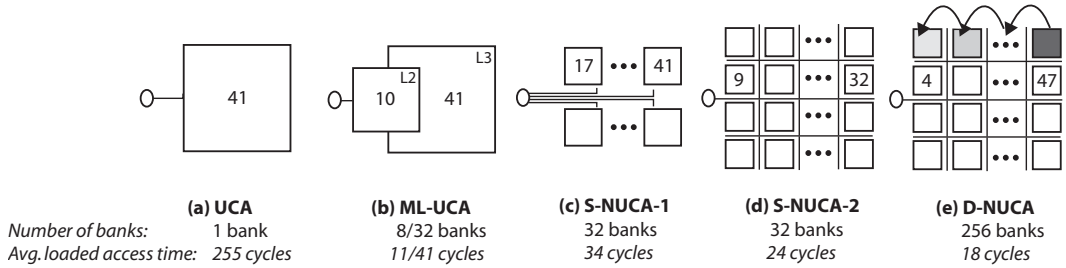


Figure 3.1: Various level-2 cache architectures.

banks are not shown in the figure. Inclusion is enforced, so a line in the smaller level implies two copies in the cache, consuming extra space.

Figure 3.1c shows an aggressively banked cache, which supports non-uniform access to the different banks without the inclusion overhead of ML-UCA. The mapping of data into banks is predetermined, based on the block index, and thus can reside in only one bank of the cache. Each bank uses a private, two-way, pipelined transmission channel to service requests. We call this statically mapped, non-uniform cache S-NUCA-1.

When the delay to route a signal across a cache is significant, increasing the number of banks can improve performance. A large bank can be subdivided into smaller banks, some of which will be closer to the cache controller, and hence faster than those farther from the cache controller. The original, larger bank was necessarily accessed at the speed of the farthest, and hence slowest, sub-bank. Increasing the number of banks, however, can increase wire and decoder area overhead. Private per-bank channels, used in S-NUCA-1, heavily restrict the number of banks that can be implemented, since the per-bank channel wires adds significant area overhead to the cache if the number of banks is large. To circumvent that limitation, we explore a static NUCA design that uses a two-dimensional switched network instead of private per-bank channels, permitting a larger number of smaller, faster

banks. This organization, called S-NUCA-2, is shown in Figure 3.1d. Figure 3.1e represents the D-NUCA organization that allows frequently used data to be migrated into closer banks to further reduce the cache hit latencies. We describe detailed mechanisms to support dynamic data migration within a cache and evaluate performance in Chapter 4.

At the end of this chapter, we show our implementation of composable secondary memory systems in the TRIPS prototype [99]. TRIPS is a novel distributed architecture that is built in 130nm ASIC technologies. The chip contains two processor cores and the 1MB on-chip secondary memory. The TRIPS secondary memory system is based on the S-NUCA-2 design. The flexibility of a switched network in S-NUCA-2 allows various memory organizations on the same cache substrate. The TRIPS secondary memory system is *composable*, meaning that it consists of multiple partitioned memory banks and each memory bank can be configured differently and be aggregated to compose various memory organizations. The possible memory organizations include a 1MB L2 cache or a 1MB scratchpad memory or any combinations between them.

3.1 Uniform Access Caches

Large modern caches are subdivided into multiple sub-banks to minimize access time. Cache modeling tools, such as Cacti [58, 124], enable fast exploration of the cache design space by automatically choosing the optimal sub-bank count, size, and orientation. To estimate the cache bank delay, we used Cacti 3.0, which accounts for capacity, sub-bank organization, area, and process technology [105].

Figure 3.2 contains an example of a Cacti-style bank, shown in the circular expanded section of one bank. The cache is modeled assuming a central pre-decoder, which drives signals to the local decoders in the sub-banks. Data are accessed at each sub-bank

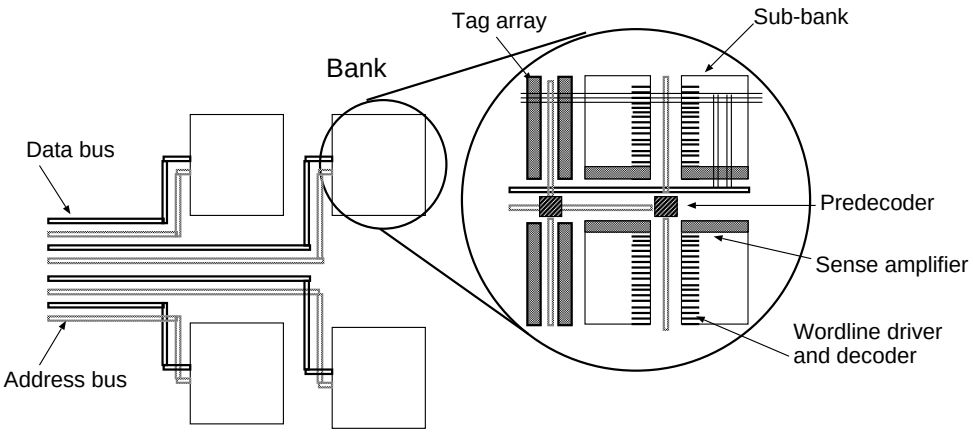


Figure 3.2: UCA and S-NUCA-1 cache design

and returned to the output drivers after passing through muxes, where the requested line is assembled and driven to the cache controller. Cacti uses an exhaustive search to choose the number and shape of sub-banks to minimize access time. Despite the use of an optimal sub-banking organization, large caches of this type perform poorly in a wire-delay-dominated process, since the delay to receive the portion of a line from the slowest of the sub-banks is large.

3.1.1 Experimental Methodology

To evaluate the effects of different cache organizations on system performance, we used Cacti to derive the access times for caches, and extended the `sim-alpha` simulator [24] to simulate different cache organizations with parameters derived from Cacti. The `sim-alpha` simulator models an Alpha 21264 core in detail [61]. We assumed that all microarchitectural parameters other than the L2 organization match those of the 21264, including issue width, fetch bandwidth, and clustering. The L1 caches we simulated are similar to those

	Phase		L2 load accesses/		Phase		L2 load accesses/
SPECINT2000	FFWD	RUN	Million instr	SPECFP2000	FFWD	RUN	Million instr
176.gcc	2.367B	300M	25,900	172.mgrid	550M	1.06B	21,000
181.mcf	5.0B	200M	260,620	177.mesa	570M	200M	2,500
197.parser	3.709B	200M	14,400	173.applu	267M	650M	43,300
253.perlbmk	5.0B	200M	26,500	179.art	2.2B	200M	136,500
256.bzip2	744M	1.0B	9,300	178.galgel	4.0B	200M	44,600
300.twolf	511M	200M	22,500	183.equake	4.459B	200M	41,100
Speech				NAS			
sphinx	6.0B	200M	54,200	cg	600M	200M	113,900
				bt	800M	650M	34,500
				sp	2.5B	200M	67,200

Table 3.1: Benchmarks used for performance experiments

of the 21264: 3-cycle access to the 64KB, 2-way set associative L1 data cache, and single-cycle access to the similarly configured L1 I-cache. All line sizes in this study were fixed at 64 bytes. In all cache experiments, we assumed that the off-chip memory controller resides near the L2 memory controller. Thus, writebacks need to be pulled out of the cache, and demand misses, when the pertinent line arrives, are injected into the cache by the L2 controller, with all contention modeled as necessary. However, we do not model any routing latency from the off-chip memory controller to the L2 cache controller.

Table 3.1 shows the benchmarks used in our experiments, chosen for their high L1 miss rates. The 16 applications include six SPEC2000 floating-point benchmarks [115], six SPEC2000 integer benchmarks, three scientific applications from the NAS suite [8], and Sphinx, a speech recognition application [70]. For each benchmark we simulated the sequence of instructions which capture the core repetitive phase of the program, determined empirically by plotting the L2 miss rates over one execution of each benchmark, and choosing the smallest subsequence that captured the recurrent behavior of the benchmark. Table 3.1 lists the number of instructions skipped to reach the phase start (FFWD) and the number of instructions simulated (RUN). Table 3.1 also shows the anticipated L2 load, list-

Tech (nm)	L2 Capacity	Num. Sub-banks	Unloaded Latency	Loaded Latency	IPC	Miss Rate
130	2MB	16	13	67.7	0.41	0.23
90	4MB	16	18	91.1	0.39	0.20
65	8MB	32	26	144.2	0.34	0.17
45	16MB	32	41	255.1	0.26	0.13

Table 3.2: Performance of UCA organizations

ing the number of L2 accesses per 1 million instructions assuming 64KB level-1 instruction and data caches. (This metric was proposed by Kessler *et al.* [62].)

3.1.2 UCA Evaluation

Table 3.2 shows the parameters and achieved instructions per cycle (IPC) of the UCA organization. For the rest of this chapter, we assume a constant L2 cache area and vary the technology generation to scale cache capacity within that area, using the SIA Roadmap [101] predictions, from 2MB of on-chip L2 at 130nm devices to 16MB at 45nm devices. In Table 3.2, the unloaded latency is the average access time (in cycles) assuming uniform bank access distribution and no contention. The loaded latency is obtained by averaging the actual L2 cache access time–including contention–across all of the benchmarks. Contention can include both *bank contention*, when a request must stall because the needed bank is busy servicing a different request, and *channel contention*, when the bank is free but the routing path to the bank is busy, delaying a request.

The reported IPCs are the harmonic mean of all IPC values across our benchmarks, and the cache configuration displayed for each capacity is the one that produced the best IPC; we varied the number and aspect ratio of sub-banks exhaustively, as well as the number of banks.

In the UCA cache, the unloaded access latencies are sufficiently high that contention could be a serious problem. Multiported cells are a poor solution for overlapping accesses in large caches, as increases in area will expand loaded access times significantly: for a 2-ported, 16MB cache at 45nm, Cacti reports a significant increase in the unloaded latency, which makes a 2-ported solution perform worse than a single-ported L2 cache. Instead of multiple physical ports per cell, we assume perfect pipelining: that all routing and logic have latches, and that a new request could be initiated at an interval determined by the maximal sub-bank delay, which is shown in column 4 of Table 3.2. We did not model the area or delay consumed by the pipeline latches, resulting in optimistic performance projections for an UCA organization.

Table 3.2 shows that, despite the aggressive cache pipelining, the loaded latency grows significantly as the cache size increases, from 68 cycles at 2MB to 255 cycles at 16MB. The best overall cache size is 2MB, at which the increases in L2 latency are subsumed by the improvement in miss rates. For larger caches, the latency increases overwhelm the continued reduction in L2 misses. While the UCA organization is inappropriate for large, wire-dominated caches, it serves as a baseline for measuring the performance improvement of more sophisticated cache organizations, described in the following section.

3.2 Static NUCA Implementations

Much performance is lost by requiring worst-case uniform access in a wire-delay dominated cache. Multiple banks can mitigate those losses, if each bank can be accessed at different speeds, proportional to the distance of the bank from the cache controller. Each bank is independently addressable, and is sized and partitioned into a locally optimal physical sub-bank organization. As before, the number and physical organization of banks and sub-banks

were chosen to maximize overall IPC, after an exhaustive exploration of the design space.

Data are statically mapped into banks, with the low-order bits of the index determining the bank. Each bank we simulate is four-way set associative. These static, non-uniform cache architectures (S-NUCA) have two advantages over the UCA organization previously described. First, accesses to banks closer to the cache controller incur lower latency. Second, accesses to different banks may proceed in parallel, reducing contention. We call these caches S-NUCA caches, since the mappings of data to banks are static, and the banks have non-uniform access times.

3.2.1 Private Channels

As shown in Figure 3.2, each addressable bank in the S-NUCA-1 organization has two private, per-bank 128-bit channels, one going in each direction. Cacti 3.0 is not suited for modeling these long transmission channels, since it uses the Rubenstein RC wire delay model [55] and assumes bit-line capacitative loading on each wire. We replaced that model with the more aggressive repeater and scaled wire model of Agarwal *et al.* for the long address and data busses to and from the banks [2].

Since banks have private channels, each bank can be accessed independently at its maximum speed. While smaller banks would provide more concurrency and a greater fidelity of non-uniform access, the numerous per-bank channels add area overhead to the array that constrains the number of banks.

When a bank conflict occurs, we model contention in two ways. A *conservative* policy assumes a simple scheduler that does not place a request on a bank channel until the previous request to that bank has completed. Bank requests may thus be initiated every $b + 2d + 3$ cycles, where b is the actual bank access time, d is the one-way transmission

Technology	L2	Num.	Unloaded latency				Conservative		Aggressive	
(nm)	size	banks	bank	min	max	avg.	Loaded	IPC	Loaded	IPC
130	2MB	16	3	7	13	10	11.3	0.54	10.0	0.55
90	4MB	32	3	9	21	15	17.3	0.56	15.3	0.57
65	8MB	32	5	12	26	19	21.9	0.61	19.3	0.63
45	16MB	32	8	17	41	29	34.2	0.59	30.2	0.62

Table 3.3: S-NUCA-1 evaluation

time on a bank's channel, and the additional 3 cycles are needed to drain the additional data packets on the channel in the case of a read request following a writeback. Since each channel is 16 bytes, and the L2 cache line size is 64 bytes, it takes 4 cycles to remove a cache line from the channel.

An *aggressive* pipelining policy assumes that a request to a bank may be initiated every $b + 3$ cycles, where b is the access latency of the bank itself. This channel model is optimistic, as we do not model the delay or area overhead of the latches necessary to have multiple requests in flight on a channel at once, although we do model the delay of the wire repeaters.

Table 3.3 shows a breakdown of the access delays for the various cache sizes and technology points: the number of banks to which independent requests can be sent simultaneously, the raw bank access delay, the minimum, average, and maximum access latency of a single request to various banks, and the average latency seen at run-time (including channel contention). We assume that the cache controller resides in the middle of one side of the bank array, so the farthest distance that must be traversed is half of one dimension and the entire other dimension. Unlike UCA, the average IPC increases as the cache sizes increases, until 8 MB. At 16MB, the large area taken by the cache causes the hit latencies to overwhelm the reduced misses, even though the access latencies grow more slowly than with an UCA organization.

As technology advances, both the access time of individual banks and the routing delay to the farthest banks increase. The bank access times for S-NUCA-1 increase from 3 cycles at 100nm to 8 cycles at 45nm because the best organization at smaller technologies uses larger banks. The overhead of the larger, slower banks is less than the delays that would be caused by the extra wires required for more numerous, smaller banks.

The greater wire delays at small technologies cause increased routing delays to the farther banks. At 130nm, the worst-case routing delay is 10 cycles. It increases steadily to reach 33 cycles at 45nm. While raw routing delays in the cache are significant, contention is less of a problem. Contention for banks and channels can be measured by subtracting the average loaded latency from the average unloaded latency in Table 3.3. The aggressive pipelining of the request transmission on the channels eliminates from 1.3 to 4.0 cycles from the conservative pipelining average loaded bank access latency, resulting in a 5% improvement in IPC at 16MB.

The ideal number of banks increases from 16 at 2MB to 32 at 4MB. At 8MB and 16MB, the ideal number of banks does not increase further, due to the area overhead of the per-bank channels, so each bank grows larger and slower as the cache size increases. That constraint prevents the S-NUCA-1 organization from exploiting the potential access fidelity of small, fast banks. In the next subsection, we describe a inter-bank network that mitigates the per-bank channel area constraint.

3.2.2 Switched Channels

Figure 3.3 shows an organization that removes most of the large number of wires resulting from per-bank channels. This organization embeds a lightweight, wormhole-routed 2-D mesh with point-to-point links in the cache, placing simple switches at each bank. Each link

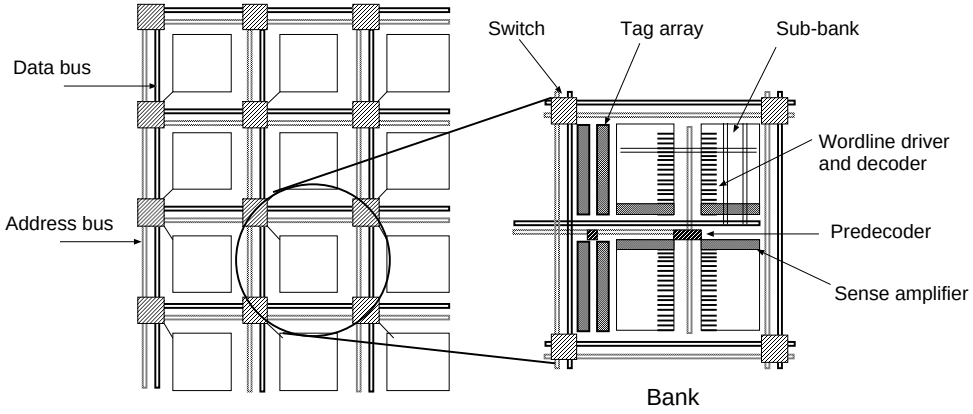


Figure 3.3: Switched NUCA design

has two separate 128-bit channels for bidirectional routing. We modeled the switch logic in HSPICE to obtain the delay for each switch and incorporate that delay into performance simulations. We again used the Agarwal *et al.* model for measuring wire delay between switches. As in the previous configurations, we assume 4-way set associative banks.

We modeled contention by implementing wormhole-routed flow control, and by simulating the mesh itself and the individual switch occupancy in detail as a part of performance simulations. In our simulations, each switch buffers 16-byte packets, and each bank contains a larger buffer to hold an entire pending request. Thus, exactly one request can be queued at a specific bank while another is being serviced. A third arrival would block the network links, buffering the third request in the network switches and delaying other requests requiring those switches. Other banks along different network paths could still be accessed in parallel, of course.

In the highest-performing bank organization presented, each bank was sized so that the routing delay along one bank was just under one cycle. We simulated switches that had buffer slots for four flits per channel, since our sensitivity analysis showed that more than

Technology (nm)	L2 Size	Num. Banks	Unloaded Latency bank	min	max	avg.	Loaded Latency	IPC	Bank Requests
130	2MB	16	3	4	11	8	9.7	0.55	17M
90	4MB	32	3	4	15	10	11.9	0.58	16M
65	8MB	32	5	6	29	18	20.6	0.62	15M
45	16MB	32	8	9	32	21	24.2	0.65	15M

Table 3.4: S-NUCA-2 performance

four slots per switch gained little additional IPC. In our 16MB S-NUCA-2 simulations, the cache incurred an average of 0.8 cycles of bank contention and 0.7 cycles of link contention in the network.

Table 3.4 shows the IPC of the S-NUCA-2 design. For 4MB and larger caches, the minimum, average, and maximum bank latencies are significantly smaller than those for S-NUCA-1. The switched network speeds up cache accesses because it consumes less area than the private, per-bank channels, resulting in a smaller array and faster access to all banks. At 45nm with 32 banks, our models indicate that the S-NUCA-1 organization's wires consume 20.9% of the bank area, whereas the S-NUCA-2 channel overhead is just 5.9% the total area of the banks.

The S-NUCA-2 cache is faster at every technology than S-NUCA-1, and furthermore at 45nm with a 16MB cache, the average loaded latency is 24.2 cycles, as opposed to 34.2 cycles for S-NUCA-1. At 16MB, that reduction in latency results in a 10% average improvement in IPC across the benchmark suite. An additional benefit from the reduced per-bank wire overhead is that larger numbers of banks are possible and desirable, as we show in the following section.

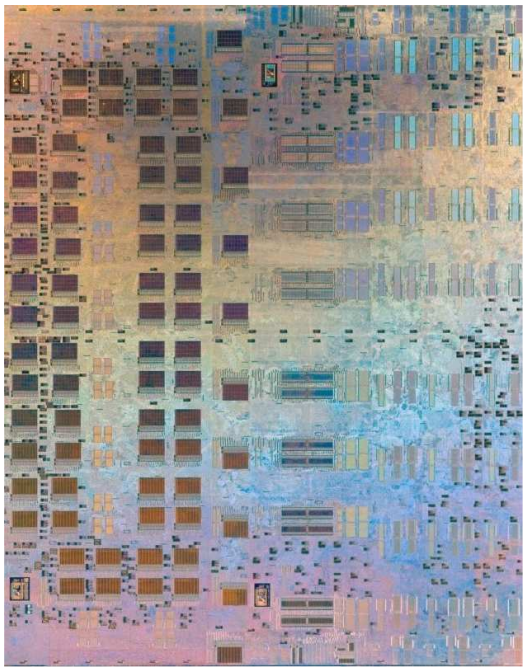


Figure 3.4: TRIPS die photo

3.3 TRIPS NUCA design

We showed that the switched static NUCA performs better than the channeled static NUCA since the switched network has less area overhead than the private, per-bank channels. In addition to the performance benefit, the configuration of the switched networks provides a variety of on-chip memory organizations on the same substrate. As a proof of concept, we implemented a 1MB switched static NUCA design in the TRIPS prototype hardware [99]. TRIPS is a novel distributed architecture which is composed of two coarse-grained processors [84] and a shared NUCA L2 cache. The prototype chip is fabricated in a 130nm IBM ASIC technology and has more than 170 million transistors [99]. Figure 3.4 shows the die photo of the TRIPS chip.

TRIPS secondary memory system: The TRIPS secondary memory system has the following five characteristics.

1. Non-uniform access latency: The TRIPS NUCA design consists of 16 64KB memory banks. The highly partitioned NUCA design is more tolerant to increasing on-chip wire delays in future technologies. Compared to conventional caches that have an uniform access latency, close cache banks from the processor can be accessed faster than cache banks that are located far from the processor.

2. High-bandwidth access: Ten pairs of 128-bit data channels allows the NUCA cache to communicate with the two TRIPS processors. At the architected frequency of 500MHz, the peak injection bandwidth is 74 GB/sec, which provides high-bandwidth data accesses for streaming applications.

3. Composibility: Each memory bank can be configured as either a L2 cache bank or an explicitly addressable scratchpad memory. Depending on applications' memory access patterns, the TRIPS NUCA design allows each memory bank to be configured differently and be aggregated to compose various memory organizations. This composable capability provides a flexibility to organize the secondary memory system as a 1MB L2 cache, 1MB on-chip physical memory (no L2 cache) and many combinations in between at the granularity of 64KB increments.

4. Configurability: The TRIPS NUCA design supports two types of cache line interleaving modes to access L2 cache - interleaved or split mode. The interleaved mode (or shared cache mode) allows a single application to better utilize a 1MB L2 cache and the on-chip network bandwidth. In split mode (or called private cache mode), each processor can use a 512KB L2 cache region privately without interfering with

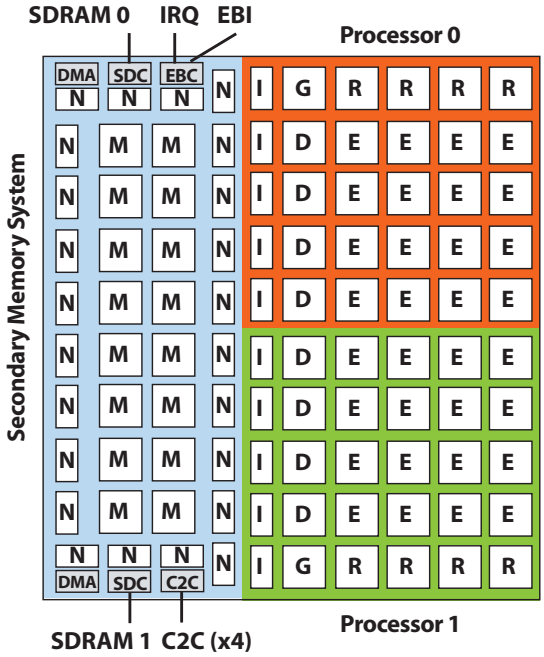


Figure 3.5: TRIPS prototype block diagram

the other assuming the OS page mapping is set up appropriately.

5. High connectivity: The On-chip network (OCN) that are embedded in the TRIPS NUCA serves as the SoC ("System on a Chip") interconnect. The TRIPS OCN provides higher connectivity than the current standardized bus design for SoC interconnects such as the AMBA bus from ARM [33]. The TRIPS OCN connects two processors, two SDRAM controllers, two DMA controllers, the External Bus Interface controller, the Chip to Chip controller, and a 1MB NUCA array.

3.3.1 TRIPS Chip Overview

Figure 3.5 shows the block diagram of the TRIPS prototype chip [99]. The TRIPS chip contains two processor cores and a 1MB NUCA array as the major components. Each of the

two processor cores is composed of five different types of tiles: one global control tile (GT), sixteen execution tiles (ET), five instruction cache tiles (IT), four data cache tiles (DT), and four register tiles (RT). A scalar operand network and multiple control networks connect all of the tiles and construct a processor core with 16-wide out-of-order issue, 64KB L1 instruction cache and 32KB of L1 data cache. In single-thread mode, a processor executes up to 1024 instructions in flight. A multi-threaded mode partitions execution resources and supports up to four different threads running concurrently on a single core. The TRIPS processor implements an Explicit Data Graph Execution (EDGE) instruction set architecture [15] that allows power-efficient exploitation of concurrency over distributed tiles.

3.3.2 TRIPS Secondary Memory Subsystem

The TRIPS secondary memory subsystem consists of forty tiles - 16 Memory Tiles (MT) and 24 Network Tiles (NT). Each tile is connected to the On-Chip Network (OCN).

Memory Tiles (MT)

As shown in Figure 3.6, a Memory Tile includes an OCN router subcomponent and a 64KB SRAM bank. The OCN router supports four different virtual channels to prevent deadlocks. Incoming packets are buffered at the input FIFO in one of five directions, North, South, East, West, or Local for a SRAM bank itself. A 4x4 crossbar switch connects each input to all possible output channels except that the input from one direction cannot be routed to the output from the same direction.

The 64KB SRAM bank can be configured as part of a L2 cache or as part of a scratchpad memory. In L2 cache mode, the SRAM bank acts as a single bank in a larger L2 cache. To track a L2 miss in flight, each Memory Tile contains a single-entry Miss

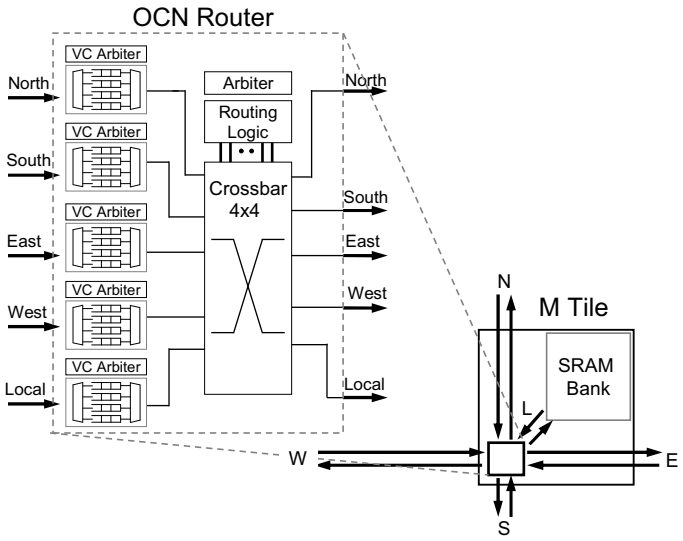


Figure 3.6: Memory tile block diagram highlighting OCN router in detail

Status Holding Register (MSHR). When configured as part of the scratchpad memory, the tag checks are turned off to allow direct data array accesses. In both L2 cache mode and scratchpad memory mode, the MT requires three cycles from receiving a request to producing the first reply packet.

Network Tiles (NT)

Figure 3.7 contains a detailed block diagram of the Network Tile. The Network Tile consists of a network router subcomponent and an address translation unit.

A network router subcomponent is similar to the one used in a Memory Tile. While the local channels in a Memory Tile are connected to the memory bank, the local channels in a Network Tile are connected to the OCN clients, such as processors, and I/O units. Another difference is that a 5x5 crossbar switch is used instead of a 4x4 crossbar to add a configuration path to modify the contents of the address translation unit.

The TRIPS OCN introduces the mapping between a physical address and a location

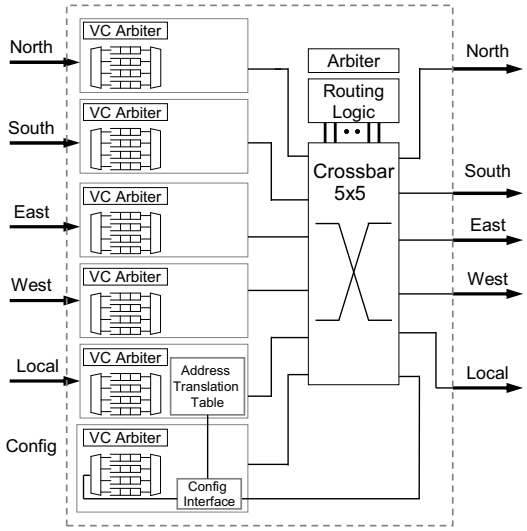


Figure 3.7: Network tile block diagram in detail

of tiles (X, Y coordinates) in the OCN to allow various memory organizations. This mapping occurs in two steps. First, addresses are mapped onto sixteen logical L2 cache bank. Second, each logical L2 cache bank is mapped to a specific MT or SDRAM controller by the address translation unit. Each entry in the address translation unit contains the X-Y coordinates of the M-tile or the SDRAM controller to which the logical L2 cache banks are mapped. The table itself is memory mapped and can be modified on-the-fly by the runtime system.

On-Chip Network (OCN)

The TRIPS OCN connects 40 tiles in a 4x10, 2D mesh. Each tile is connected with each other using a pair of 128-bit data links. The OCN is a Y-X dimension-order, wormhole-routed network with credit-based flow control, meaning that a sender maintains a count of the number of empty buffers in a receiver and a credit is sent back to the sender whenever the receiver's buffer gets emptied. Packets travel on the following four virtual channels to

prevent potential deadlock scenarios.

- Primary Reply (P1) - Replies to network clients (first priority)
- Secondary Reply (P2) - L2 cache fill and spill replies (second priority)
- Secondary Request (Q2) - L2 cache fill and spill requests (third priority)
- Primary Request (Q1) - Requests from network clients (fourth priority)

The OCN supports read, write, and swap transactions. Each transaction consists of a request and a reply. The packets for requests and replies range in size from 16 byte to 80 bytes long broken up into one to five 16 byte flits. The first flit is called the "header flit", which contains the transaction type, size, locations for source and destination tiles, and address information. The remaining flits are the payload, which carry from one to 64 bytes of data. More detailed information on the TRIPS OCN can be found elsewhere [39].

3.3.3 Composable Secondary Memory Organization

The TRIPS secondary memory system is *composable* in the sense that it consists of multiple partitioned memory banks, which can be configured differently and aggregated to compose various memory organizations. Figure 3.8 shows various possible secondary memory organizations: A 1MB L2 cache (Figure 3.8a), a 1MB scratchpad memory (Figure 3.8b), and a 512KB cache and a 512KB scratchpad memory (Figure 3.8c). Any combinations in between a 1MB L2 cache and 1MB scrachpad memory are possible at the granularity of 64KB increments.

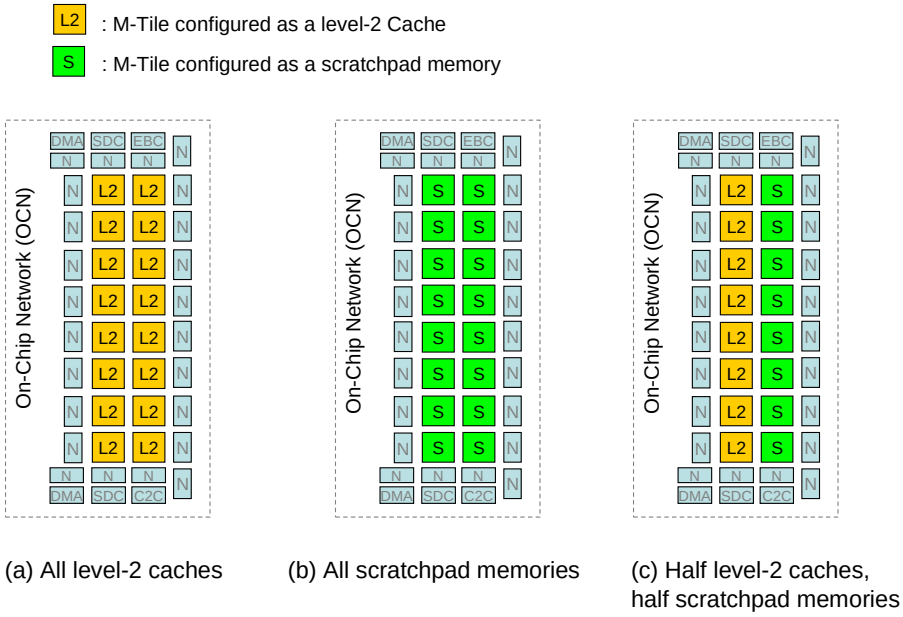


Figure 3.8: Various memory organizations in the TRIPS secondary memory system

Flexible Memory Organization

Composable flexibility comes from remapping from a logical tile location, which is fixed by a physical address into any tile location of 16 Memory Tiles or the two SDRAM controllers. When a Memory Tile is configured as scratchpad memory, the L2 requests to the corresponding Memory Tile are redirected to other Memory Tiles in L2 mode or directly to the SDRAM controller. Therefore, when all Memory Tiles are configured as scrachpad memory, all L2 traffic should be routed directly to the SDRAM controllers. These configurable mappings are effected by modifying the address translation table in the Network Tiles. Before any reconfiguration occurs, all in-flight OCN traffic must be drained and the participating Memory Tiles must be flushed. System software may then modify the address

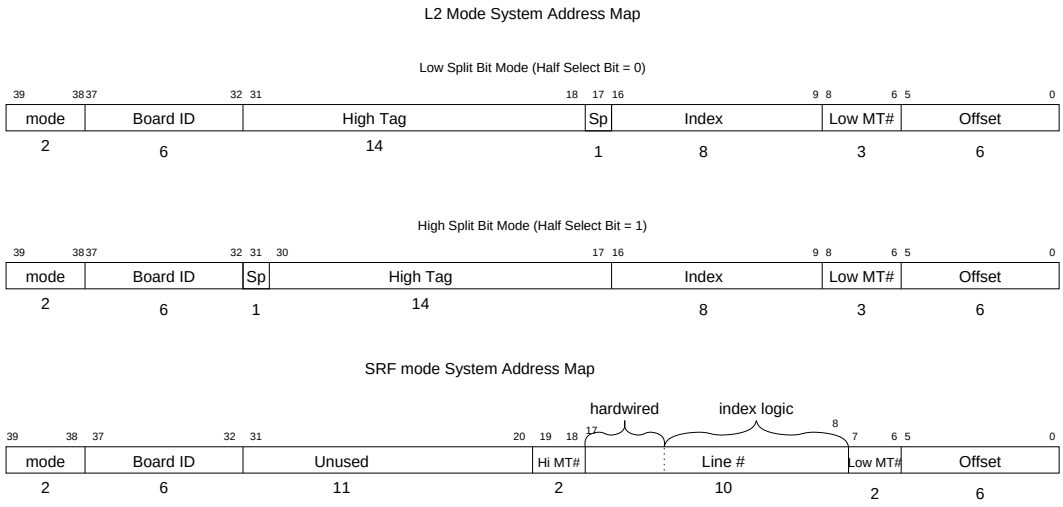


Figure 3.9: TRIPS OCN 40-bit address field composition

translation tables in all Network Tiles and resume execution.

Flexible Cache Interleaving

The TRIPS NUCA design supports two types of cache line interleaving modes to access the L2 cache - interleaved or split mode. Figure 3.9 shows how the 40-bit physical address is interpreted in the different modes. The “Split bit” (called SP) represents whether the address is mapped to one of top eight Memory Tiles or one of bottom eight Memory Tiles. In split mode, the “SP” bit is located at the 31st bit in an address, meaning that the entire 4GB address region in the chip is split into two contiguous 2GB regions. In this mode, the top eight Memory Tiles are mapped to the first 2GB region and the bottom eight Memory Tiles are mapped to the next 2GB region. Assuming the operating system allocates pages into one of two contiguous 2GB regions, each processor can use a 512KB L2 cache region privately without interfering with the other. Therefore, the split mode can be considered a ”private

cache mode". In interleaved mode, the "SP" bit is located at the 17th bit in a address, which allows L2 requests from both processors to be more evenly distributed among all sixteen Memory Tiles. Since the interleaved mode lets applications of each processor fully utilize a 1MB L2 cache and on-chip network bandwidth, the interleaved mode is also called a "shared cache mode".

3.3.4 Network Performance Evaluations

As a preliminary evaluation of the TRIPS NUCA design, we wrote a simulator called *tsim_ocn* that simulates the behavior of the L2 cache and the on-chip network at a per-cycle level. We use two different types of synthetic statistical loads to measure the maximum throughput and the average latency of the switched network that are embedded in the TRIPS NUCA design. In addition, this network evaluation provided information in determining design parameters including the number of SDRAM controller and the FIFO depth in each router.

There are five parameters that we varied to measuring OCN performance:

- Request rate: The request rate represented in the x-axis in both the throughput and latency graph can be used to estimate the ideal throughput.
- Traffic pattern: The "uniform random traffic" pattern is one of the most commonly used traffic in network evaluation [22]. Requesters distribute requests evenly to all possible and randomly chosen destinations. We use this pattern for evaluating the TRIPS processor that is configured to maximize ILP. Another traffic pattern is the "neighbor traffic" pattern. The "neighbor traffic" pattern is used for evaluating the TRIPS processor where the Memory Tiles that are attached to processors are configured as scratchpad memory. In the "neighbor traffic" pattern, a requester chooses

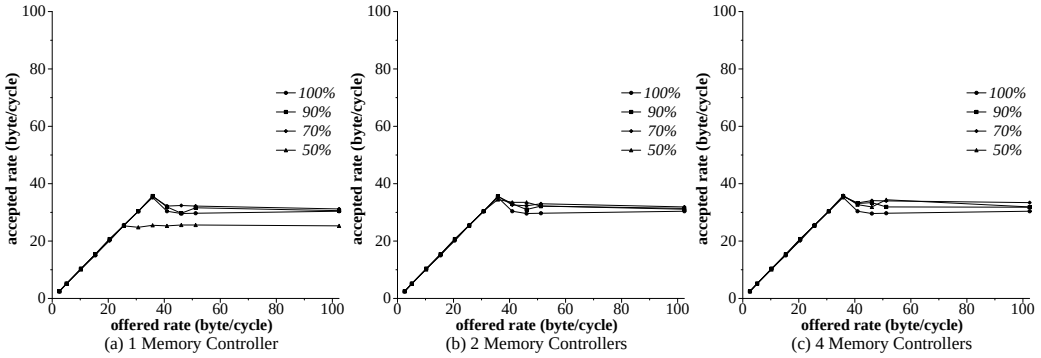


Figure 3.10: Throughput with uniform random traffic

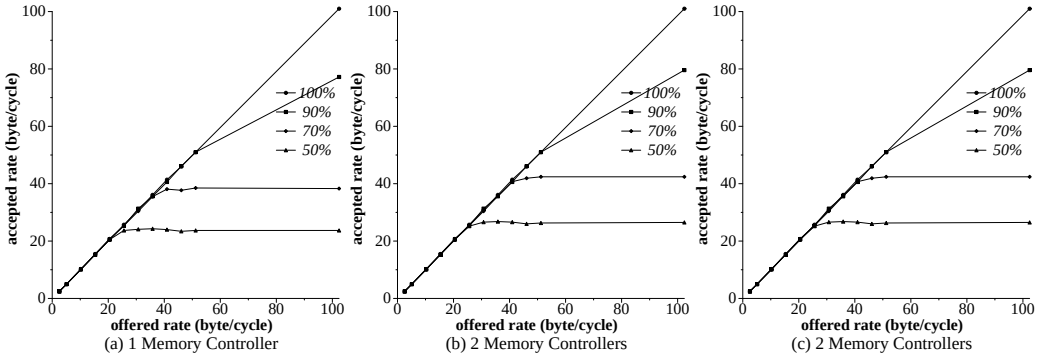
the destination between two Memory Tiles that are located in the same row as the requester.

- Hit ratio in a Memory Tile: A cache hit ratio in each Memory Tile is varied from 100%, 90%, 70% to 50%.

- Number of SDRAM controllers: The number of SDRAM controllers is varied from one, two to four.

Throughput

Each Figure 3.10a, b and c shows how the OCN throughput varies by changing the number of memory controllers among one, two and four. Figure 3.10a, b and c plot the accepted throughput as a function of offered traffic by varying a cache hit ratio among 100%, 90%, 70%, and 50%. In Figure 3.10a, the accepted traffic is increased up to 36 byte/cycle, then saturates at the 100%, 90%, 70% hit ratio. However, at the 50% hit ratio, the accepted traffic is more quickly saturated into 25 byte/cycle than other hit ratios. In both the Figure 3.10b and c, the accepted traffic at the the 50% hit ratio shows the same peak throughput as



(a) 1 Memory Controller (b) 2 Memory Controllers (c) 2 Memory Controllers

Figure 3.11: Throughput with the neighbor traffic

the rest of the hit ratios. Also, Figure 3.10b and c are identical. Considering extra off-chip requests and logic complexity (pin count), two memory controllers seems to be a reasonable compromise. To summarize, the TRIPS OCN can provide the peak throughput up to 36 byte/cycle.

Figure 3.11 shows the OCN throughput with the "neighbor traffic," which is when the nearest Memory Tile services load requests from each requester that is located in the same row. This distribution assumes the case (1) when Memory Tiles are configured as software managed memories (100% hit rate) or (2) when all cache accesses are serviced by the nearest Memory Tile. The latter case corresponds to the ideal case when dynamic data migration (described in the next chapter) works.

Figure 3.11 shows that the OCN can sustain its obtainable peak throughput (102.4 byte per cycle) with this request pattern. Interestingly, the accepted traffic is saturated very quickly at the 90%, 70%, and 50% hit ratios, and the peak throughput at the 50% hit ratio is less than that in the uniform random traffic. This phenomenon arises because each Memory Tile has only single-entry MSHR and most requests are stalled from the previous miss

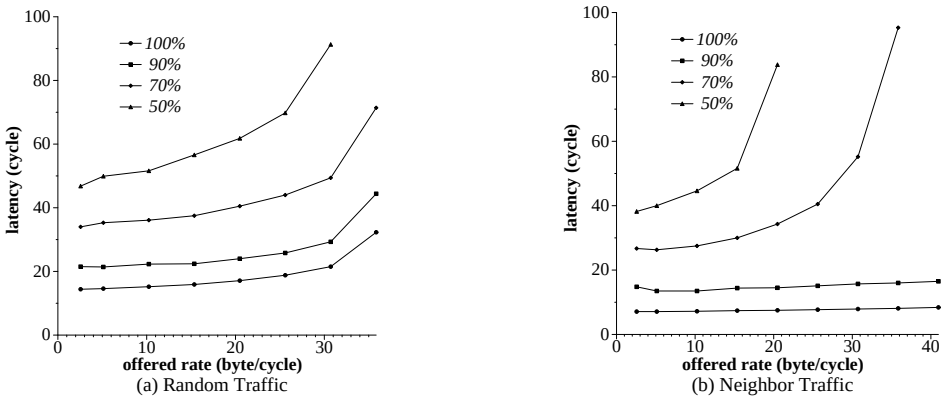


Figure 3.12: TRIPS S-NUCA cache hit latency

Cache Hit Ration in Memory Tiles	Random Traffic	Neighbor Traffic
100%	15.9 (cycles)	7.4 (cycles)
90%	22.4 (cycles)	14.4 (cycles)
70%	37.5 (cycles)	30.0 (cycles)

Table 3.5: Average L2 cache access time in TRIPS (with synthetic traffic)

request. Since the random traffic is spread more uniformly across all Memory Tiles, the buffer in each router can hold more requests than the neighbor traffic.

Latency

Figures 3.12a and b show the OCN latency with each of the random and neighbor traffic patterns. In this experiment, we define the OCN latency as the time elapsed from when the request header flit is injected into the OCN until the reply header flit is received by the requester when a request makes a cache hit on a Memory Tile. The OCN latency can thus be considered to be the average L2 hit latency in the TRIPS processor. Initially, the OCN latency gradually increases as the offered traffic grows. If the offered traffic exceedes the

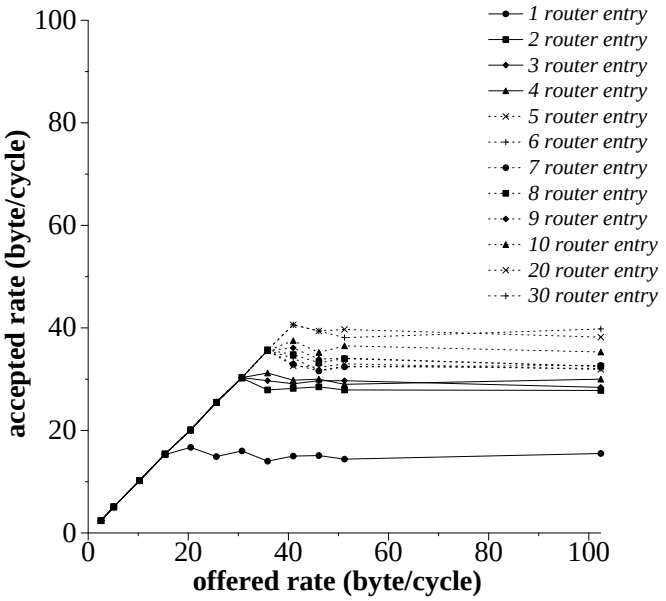


Figure 3.13: Throughput with the various FIFO depth

peak throughput, the OCN is saturated with packets and the latency goes up exponentially.

The L2 cache access time in the TRIPS design varies depending on the number of requests, traffic patterns, and hit ratios (since stalls caused by the limited number of MSHRs per Memory Tiles increase when a L2 miss ratio increases). Table 3.5 shows the TRIPS L2 cache access time measured under the various configurations.

Determining the FIFO depth

To find the optimal FIFO depth in each router, we varied the number of FIFO entries and measured achieved throughput. Figure 3.13 shows the changes in throughput when the number of FIFO entries per virtual channel increases from one to 30. There are two significant changes when the number of entries increase from one to two and four to five. After

five, no additional gains can be found until 20 entries, which results from the fact that one OCN packet consists of five flits.

3.4 Summary

To handle the problem of growing wire delays in future larger level-2 caches, we evaluated several new designs that treat a L2 cache as a network of banks and facilitates non-uniform cache accesses to different physical regions. In this chapter, we evaluated cache designs that consist of multiple independent banks connected by either private per-bank channels or a wormhole-routed 2-D switched mesh network. We compared both cache designs with a traditional cache (called Uniform Cache Architecture or UCA) and showed that the UCA design would perform poorly due to internal wire delays and a restricted number of ports.

We also showed that an embedded mesh network performs better than per-bank private channels since the switched network takes less area than the per-bank private channels. On top of the performance benefits, the configurable nature of switched networks allows various memory organizations on the same cache substrate.

As a proof of concept, we implemented a composable secondary memory system in the TRIPS prototype with S-NUCA-2 organization. The TRIPS secondary memory system is *composable*, meaning that it consists of multiple partitioned memory bank and each memory bank can be configured differently and aggregated to form various memory organizations. The possible memory organizations include a 1MB L2 cache or a 1MB scratchpad memory or any combinations between them.

Finally, for future composable on-chip memory designs, an interesting question is determining the size of composition units (memory banks) across various cache capacities and technologies. In this dissertation, we showed that growing cache capacity at future

technologies increases the size of a composition unit to keep the overall average hop count modest between the processor and the memory banks. However, even at the same technology, a designer must consider the following factors to find the right size of the composition unit. While a smaller-sized composition unit supports more flexibility to provide various memory organizations and decreases the wire delay between hops, the area overhead of composability increases. This dissertation showed a composable secondary memory system in the TRIPS prototype using a 64KB memory bank as a composition unit at 130 nm ASIC technology. Even though a switched network consumes less area than private channels, the TRIPS implementation shows that 13% of the area of the secondary memory system is devoted to routers. These routers would not be required for traditional non-composable memory systems. In future technologies, the size of a composition unit should be determined by considering the overall hop counts, the area overhead, and the wire delay between hops.

Chapter 4

Dynamically Mapped Composable Memories

In Chapter 3, we proposed a composable cache substrate that consists of multiple cache banks and each bank is connected by a switched fabric. Cache lines are statically mapped into banks, meaning that the low-order bits of each address determine the bank, and the mapping between an address and the bank does not change dynamically. In this chapter, we show how to exploit future cache access non-uniformity by automatically placing frequently accessed data in closer (faster) banks and less important–yet still cached–data in farther banks.

By providing dynamic mapping and migration of data to banks, we show policies that service most requests by the fastest bank. Using the switched network, data is gradually promoted to faster banks based on access frequency. This promotion is enabled by spreading sets across multiple banks, where each bank forms one way of a set. Thus, cache lines in closer ways can be accessed faster than lines in farther ways. This dynamic non-uniform

scheme is called D-NUCA.

In the first half of this chapter, we investigate the performance effects of dynamic data migration within a L2 cache in the context of uniprocessors. We then extend the concept of non-uniform cache access architectures to emerging chip-multiprocessors.

4.1 Uniprocessor D-NUCA

The D-NUCA organization uses the same cache substrate as S-NUCA-2; multiple cache banks are connected by a switched network. On top of the S-NUCA-2 substrate, the D-NUCA organization implements a number of hardware policies regarding where to place data after data returns from memory, how to migrate data, and how to search for data. With proper placement and migration policies, D-NUCA enables the cache to place frequently accessed blocks in the banks close to the CPU and less frequently accessed blocks in the banks that are far away from the CPU. We first explore different policies to find the best performing policy for placing and migrating data. Then, we compare the D-NUCA organization to the S-NUCA organizations and the conventional multi-level hierarchy cache organization (or ML-UCA). We show that a D-NUCA cache achieves the highest IPC across diverse applications, because it adapts to the working set of each application and moves the working set into the banks closest to the processor.

4.1.1 Policy Exploration

We evaluate a number of hardware policies that migrate data among the banks to reduce average L2 cache access time and improve overall performance. For these policies, we answer three important questions about the management of data in the cache: (1) *mapping*: how the data are mapped to the banks, and in which banks a datum can reside, (2) *search*: how the

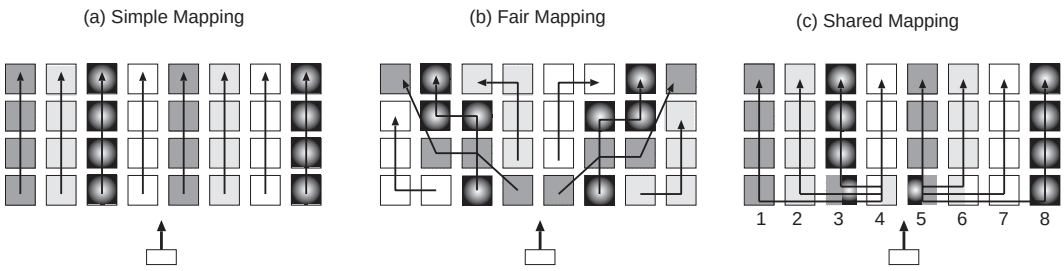


Figure 4.1: Mapping bank sets to banks in D-NUCA

set of possible locations are searched to find a line, (3) *movement*: under what conditions the data should be migrated from one bank to another. We explore these questions in each of the following subsections.

Logical to Physical Cache Mapping

A large number of banks provides substantial flexibility for mapping lines to banks. At one extreme are the S-NUCA strategies, in which a line of data can only be mapped to a single statically determined bank. At the other extreme, a line could be mapped into any cache bank. While the latter approach maximizes placement flexibility, the overhead of locating the line is larger because each bank must be searched, either through a centralized tag store or by broadcasting the tags to all of the banks.

We explore an intermediate solution called *spread sets* which treats the multibanked cache as a set-associative structure, each set is spread across multiple banks, and each bank holds a subset of the "ways" of the set. The collection of banks used to implement this associativity is called a *bank set* and the number of banks in the set, multiplied by the associativity of each bank, corresponds to the associativity.

A cache can be comprised of multiple bank sets. For example, as shown in Figure 4.1a, a cache array with 32 banks could be organized as a four-way set-associative cache, with eight bank sets, each consisting of four cache banks. To check for a hit in a spread-set cache, the pertinent tag in each of the four banks of the bank set must be checked. Note that the primary distinction between this organization and a traditional set-associative cache is that the different associative ways have different access latencies.

We evaluate the following three methods of allocating bank sets to banks: *simple mapping*, *fair mapping*, and *shared mapping*. With the simple mapping, shown in Figure 4.1a, each column of banks in the cache becomes a bank set, and all banks within a column comprise the set-associative ways. Thus, the cache may be searched for a line by first selecting the bank column, selecting the set within the column, and finally performing a tag match on banks within that column of the cache. The two drawbacks of this scheme are that the number of rows may not correspond to the number of desired ways in each bank set, and that latencies to access all bank sets are not the same; some bank sets will be faster than others, since some rows are closer to the cache controller than others.

Figure 4.1b shows the *fair mapping* policy, which addresses both problems of the simple mapping policy at the cost of additional complexity. The mapping of sets to the physical banks is indicated with the arrows and shading in the diagram. With this model, banks are allocated to bank sets so that the average access time across all bank sets is equalized. We do not present results for this policy, but describe it for completeness. The advantage of the fair mapping policy is an approximately equal average access time for each bank set. The disadvantage is a more complex routing path from bank to bank within a set, causing potentially longer routing latencies and more contention in the network.

The *shared mapping* policy, shown in Figure 4.1c, attempts to provide fastest-bank

access to all bank sets by sharing the closest banks among multiple bank sets. This policy requires that if n bank sets share a single bank, then all banks in the cache are n-way set associative. Otherwise, a swap from a solely owned bank into a shared bank could result in a line that cannot be placed into the solely owned bank, since the shared bank has fewer sets than the non-shared bank. We allow a maximum of two bank sets to share a bank. Each of the $n/2$ farthest bank sets shares half of the closest bank for one of the closest $n/2$ bank sets. This policy results in some bank sets having a slightly higher bank associativity than the others, which can offset the slightly increased average access latency to that bank set. That strategy is illustrated in Figure 4.1c, in which the bottom bank of column 3 caches lines from columns 1 and 3, the bottom bank of column 4 caches lines from columns 2 and 4, and so on. In this example the farthest four (1, 2, 7, and 8) of the eight bank sets share the closest banks of the closest four (3, 4, 5, and 6).

Locating Cached Lines

Searching for a line among a bank set can be done with two distinct policies. The first is *incremental search*, in which the banks are searched in order starting from the closest bank until the requested line is found or a miss occurs in the last bank. This policy minimizes the number of messages in the cache network and keeps energy consumption low, since fewer banks are accessed while checking for a hit, at the cost of reduced performance.

The second policy is called *multicast search*, in which the requested address is multicast to some or all of the banks in the requested bank set. Lookups proceed roughly in parallel, but at different actual times due to routing delays through the network. This scheme offers higher performance at the cost of increased energy consumption and network contention, since hits to banks far from the processor will be serviced faster than in the

incremental search policy. One potential performance drawback to multicast search is that the extra address bandwidth consumed as the address is routed to each bank may slow other accesses.

Hybrid intermediate policies are possible, such as *limited multicast*, in which the first M of the N banks in a bank set are searched in parallel, followed by an incremental search of the rest. Most of the hits will thus be serviced by a fast lookup, but the energy and network bandwidth consumed by accessing all of the ways at once will be avoided. Another hybrid policy is *partitioned multicast*, in which the bank set is broken down into subsets of banks. Each subset is searched iteratively, but the members of each subset are searched in parallel, similar to a multi-level, set-associative cache.

Partial-Tag Predictive (PTP) Search

A distributed cache array, in which the tags are distributed with the banks, creates two new challenges. First, many banks may need to be searched to find a line on a cache hit. Second, if the line is not in the cache, the slowest bank determines the time necessary to resolve that the request is a miss. The miss resolution time thus grows as the number of banks in the bank set increases. While the incremental search policy can reduce the number of bank lookups, the serialized tag lookup time increases both the hit latency and the miss resolution time.

We applied the idea of the *partial tag comparison* proposed by Kessler et al. [63] to reduce both the number of bank lookups and the miss resolution time. The D-NUCA policy using partial tag comparisons, which we call *partial-tag predictive (PTP) search*, stores the partial tag bits into a PTP search array located in the cache controller.

We evaluated two PTP search policies: *ss-performance* and *ss-energy*. In the *ss-*

performance policy, the cache array is searched as in previous policies. However, in parallel, the stored partial tag bits are compared with the corresponding bits of the requested tag, and if no matches occur, the miss processing is commenced early. In this policy, the PTP search array must contain enough of the tag bits per line to make the possibility of *false hits* low, so that upon a miss, accidental partial matches of cached tags to the requested tag are infrequent. We typically cached 6 bits from each tag, balancing the probability of incurring a false hit with the access latency to the PTP search array.

In the *ss-energy* policy, the partial tag comparison is used to reduce the number of banks that are searched upon a miss. Since the PTP search array takes multiple cycles (typically four to six) to access, serializing the PTP search array access before any cache access would significantly reduce performance. As an optimization, we allowed the access of the closest bank to proceed in parallel with the PTP search array access. After that access, if a hit in the closest bank does not occur, all other banks for which the partial tag comparison was successful are searched in parallel.

Dynamic Movement of Lines

Since the goal of the dynamic NUCA approach is to maximize the number of hits in the closest banks, a desirable policy would be to use LRU ordering to order the lines in the bank sets, with the closest bank holding the MRU line, second closest holding second most-recently used. The problem with strictly maintaining the LRU ordering is that most accesses would result in heavy movement of lines among banks. In a traditional cache, the LRU state bits are adjusted to reflect the access history of the lines, but the tags and data of the lines are not moved. In an n-way spread set, however, an access to the LRU line could result in n copy operations. Practical policies must balance the increased contention and power

consumption of copying with the benefits expected from bank set ordering.

We use *generational promotion* to reduce the amount of copying required by a pure LRU mapping, while still approximating an LRU list mapped onto the physical topology of a bank set. Generational replacement was proposed by Hallnor et al. for making replacement decisions in a software-managed UCA called the Indirect Index Cache [42]. We found that the best migration policy is that, when a hit occurs to a cache line, it is swapped with the line in the bank that is the next closest to the cache controller. Heavily used lines will thus migrate toward close, fast banks, whereas infrequently used lines will be demoted into farther, slower banks.

A D-NUCA policy must determine the placement of an incoming block resulting from a cache miss. A replacement may be loaded close to the processor, displacing an important block. The replacement may be loaded in a distant bank, in which case an important block would require several accesses before it is eventually migrated to the fastest banks. Another policy decision involves what to do with a victim upon a replacement; the two polices we evaluated were one in which the victim is evicted from the cache (a *zero-copy* policy), and one in which the victim is moved to a lower-priority bank, replacing a less important line farther from the controller (*one-copy* policy).

D-NUCA Policies

The policies we explore for D-NUCA consist of four major components: (1) **Mapping:** simple or shared. (2) **Search:** multicast, incremental, or combination. We restrict the combined policies such that a block set is partitioned into just two groups, which may then each vary in size (number of blocks) and the method of access (incremental or multicast). (3) **Promotion:** described by *promotion distance*, measured in cache banks, and *promotion*

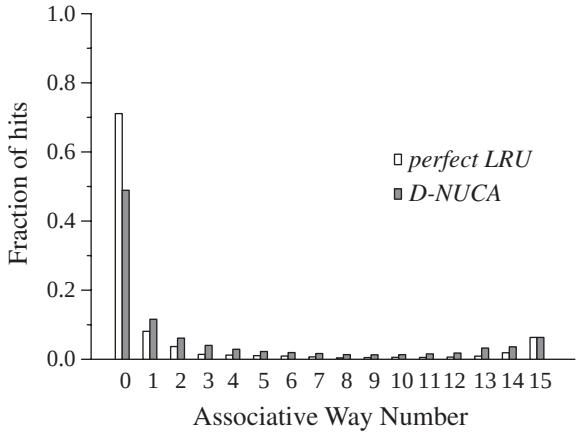


Figure 4.2: Way distribution of cache hits

trigger, measured in number of hits to a bank before a promotion occurs. (4) **Insertion:** identifies the location to place an incoming block and what to do with the block it replaces (*zero copy* or *one copy* policies).

Our simple, baseline configuration uses simple mapping, multicast search, one-bank promotion on each hit, and a replacement policy that chooses the block in the slowest bank as the victim upon a miss. To examine how effectively this replacement policy compares to pure LRU, we measured the distribution of accesses across the sets for a traditional 16-way set associative cache and a corresponding 16MB, D-NUCA cache with an 16-way bank set. Figure 4.2 shows the distribution of hits to the various sets for each cache, averaged across the benchmark suite. For both caches, most hits are concentrated in the first two ways of each set. These results are consistent with the results originally shown by So and Rechtschaffen [111], which showed that more than 90% of cache hits were to the most recently used ways in a four-way set associative cache. So and Rechtschaffen noted that a

Technology (nm)	L2 Size	Bank org. (rows x sets)	Unloaded Latency Bank	min	max	avg.	Loaded avg.	IPC	Miss Rate	Bank Accesses/Set
130	2MB	4x4	3	4	11	8	8.4	0.57	0.23	73M
90	4MB	8x4	3	4	15	10	10.0	0.63	0.19	72M
65	8MB	16x8	3	4	31	18	15.2	0.67	0.15	138M
45	16MB	16x16	3	3	47	25	18.3	0.71	0.11	266M

Table 4.1: D-NUCA base performance

transient increase in non-MRU accesses could be used to mark phase transitions, in which a new working set was being loaded.

The D-NUCA accesses are still concentrated in the banks corresponding to the most recently used bank. However, the experiments demonstrate a larger number of accesses to the non-MRU ways, since each line must gradually traverse the spread set to reach the fastest bank, instead of being instantly loaded into the MRU position, as in a conventional cache.

4.1.2 Performance Evaluation

Table 4.1 shows the performance of the baseline D-NUCA configuration, which uses the simple mapping, multicast search, tail insertion, and single-bank promotion upon each hit. As with all other experiments, for each capacity, we chose the bank and network organization that maximized overall performance. Since the shared mapping policy requires 2-way associative banks, all banks in each experiment were 2-way set associative.

As the capacities increase with the smaller technologies, from 2MB to 16MB, the average D-NUCA access latency increases by 10 cycles, from 8.4 to 18.3. The ML-UCA and S-NUCA designs incur higher average latencies at 16MB, which are 22.3 and 30.4 cycles, respectively. Data migration enables the low average latency at 16MB, which, despite the cache's larger capacity and smaller device sizes, is *less* than the average hit latency for

Policy	Av. Lat.	IPC	Miss Rate	Bank Access	Policy	Av. lat.	IPC	Miss Rate	Bank Access
Search					*Promotion*				
Incremental	24.9	0.65	0.114	89M	1-bank/2-hit	18.5	0.71	0.115	259M
2 mcast + 14 inc	23.8	0.65	0.113	96M	2-bank/1-hit	17.7	0.71	0.114	266M
2 inc + 14 mcast	20.1	0.70	0.114	127M	2-bank/2-hit	18.3	0.71	0.115	259M
2 mcast + 14 mcast	19.1	0.71	0.113	134M	*Eviction (random eviction, 1 copy)*				
Mapping					insert head	15.5	0.70	0.117	267M
Fast shared	16.6	0.73	0.119	266M	insert middle	16.6	0.70	0.114	267M
Baseline: simple map, multicast, 1-bank/1-hit, insert at tail						18.3	0.71	0.114	266M

Table 4.2: D-NUCA policy space evaluation

the 130nm, 2MB UCA organization.

At smaller capacities such as 2MB, the base D-NUCA policy shows small ($\sim$4%) IPC gains over the best of the S-NUCA and UCA organizations. The disparity grows as the cache size increases, with the base 16MB D-NUCA organization showing an average 9% IPC boost over the best-performing S-NUCA organization.

Table 4.1 also lists miss rates and the total number of accesses to individual cache banks. The number of bank accesses decreases as the cache size grows because the miss rate decreases and fewer cache fills and evictions are required. However, at 8MB and 16MB the number of bank accesses increase significantly because the multicast policy generates substantially more cache bank accesses when the number of banks in each bank set doubles from four to eight at 8MB, and again from eight to 16 at 16MB. Incremental search policies reduce the number of bank accesses at the cost of occasional added hit latency and slightly reduced IPC.

Policy Evaluation

Table 4.2 shows the IPC effects of using the baseline configuration and adjusting each policy independently. Changing the mapping function from simple to fair reduces IPC due to contention in the switched network, even though unloaded latencies are lower. Shifting from the baseline multicast to a purely incremental search policy substantially reduces the number of bank accesses by 67%. However, even though most data are found in one of the first two banks, the incremental policy increases the average access latency from 18.3 cycles to 24.9 cycles and reduces IPC by 10%. The hybrid policies (such as multicast-2/multicast-14) gain back most of the loss in access latency (19.1 cycles) and nearly all of the IPC, while still eliminating a great many of the extra bank accesses.

The data promotion policy, in which blocks may be promoted only after multiple hits, or blocks may be promoted multiple banks on a hit, has little effect on overall IPC, as seen by the three experiments in Table 4.2. The best eviction policy is as shown in the base case, replacing the block at the tail. By replacing the head, and copying it into a random, lower-priority set, the average hit time is reduced, but the increase in misses (11.4% to 11.7%) offsets the gains from the lower access latencies.

While the baseline policy is among the best-performing, using the 2 multicast/14-multicast hybrid look-up reduces the number of bank accesses to 134 million (a 50% reduction) with a mere 1% drop in IPC. However, the number of bank accesses is still significantly higher than any of the static cache organizations. Table 4.3, shows the efficacy of the PTP search policy at improving IPC *and* reducing bank accesses. We computed the size and access width of the different possible PTP search configurations, and model their access latencies accurately using Cacti.

By initiating misses early, the SS-performance policy results in a 8% IPC gain, at

the cost of an additional 1-2% area (a 224KB PTP search tag array). In the SS-energy policy, a reduction of 85% of the bank lookups can be achieved by caching seven bits of tag per line, with a 6% IPC gain over the base D-NUCA configuration. Coupling the SS-energy policy with the shared mapping policy results in a slightly larger tag array due to the increased associativity, so we reduced the PTP search tag width to six bits to keep the array access time at five cycles. However, that policy results in what we believe our best policy to be: 47M bank accesses on average, and a mean IPC of 0.75. The last two rows of Table 4.3 shows two upper bounds on IPC. The first upper bound row shows the mean IPC that would result if all accesses hit in the closest bank with no contention, costing three cycles. The second row shows the same metric, but with early initiation of misses provided by the PTP search array. The highest IPC achievable was 0.89, which is 16% better than the highest-performing D-NUCA configuration. We call the policy of SS-energy with the shared mapping the "best" D-NUCA policy DN-best, since it balances high performance with a relatively small number of bank accesses. The upper bound is 19% than the DN-best policy.

Comparison to ML-UCA

Multi-level hierarchies permit a subset of frequently used data to migrate to a smaller, closer structure, just as does a D-NUCA design, but at a coarser grain than individual banks. We compared the NUCA schemes with a two-level hierarchy (L2 and L3), called ML-UCA. We modeled the L2/L3 hierarchy as follows: we assumed that both levels were aggressively pipelined and banked UCA structures. We also assumed that the L3 had the same size as the comparable NUCA cache, and chose the L2 size and L3 organization that maximized the overall IPC. The ML-UCA organization thus consumes more area than the single-level

Configuration	Loaded Latency	Average IPC	Miss Rate	Bank Accesses	Tag Bits	Search Array
Base D-NUCA	18.3	0.71	0.113	266M	-	—
SS-performance	18.3	0.76	0.113	253M	7	224KB
SS-energy	20.8	0.74	0.113	40M	7	224KB
SS-performance + shared bank	16.6	0.77	0.119	266M	6	216KB
SS-energy + shared bank	19.2	0.75	0.119	47M	6	216KB
Upper bound	3.0	0.83	0.114	—	-	—
Upper bound + SS-performance	3.0	0.89	0.114	—	7	224KB

Table 4.3: Performance of D-NUCA with PTP search

Technology (nm)	L2/L3 Size	Num. Banks	Unloaded Latency	Loaded Latency	ML-UCA IPC	DN-best IPC
130	512KB/2MB	4/16	6/13	7.1/13.2	0.55	0.58
90	512KB/4MB	4/32	7/21	8.0/21.1	0.57	0.63
65	1MB/8MB	8/32	9/26	9.9/26.1	0.64	0.70
45	1MB/16MB	8/32	10/41	10.9/41.3	0.64	0.75

Table 4.4: Performance of an L2/L3 Hierarchy

L2 caches, and has a greater total capacity of bits. In addition, we assumed no additional routing penalty to get from the L2 to the L3 upon an L2 miss, essentially assuming that the L2 and the L3 reside in the same space, making the multi-level model optimistic.

Table 4.4 compares the IPC of the ideal two-level ML-UCA with a D-NUCA. In addition to the optimistic ML-UCA assumptions listed above, we assumed that the two levels were searched in parallel upon every access[1]. The IPC of the two schemes is roughly comparable at 2MB, but diverges as the caches grow larger. At 16MB, the overall IPC is 17% higher with DN-best than with the ML-UCA, since many of the applications have working sets greater than 2MB, incurring unnecessary misses, and some have working sets smaller than 2MB, rendering the ML-UCA L2 too slow.

[1]The IPC of an ML-UCA design was 4% to 5% worse when the L2 and L3 were searched serially instead of in parallel.

Tech. model	Num. banks	Configuration	Loaded latency	Average IPC	Miss rate	Bank accesses
SIA 1999	32	S-NUCA1	21.9	0.68	0.13	15M
	64	Shared bank D-NUCA	12.5	0.78	0.12	144M
		SS-energy + shared bank	15.6	0.78	0.12	36M
SIA 2001	32	S-NUCA1	30.2	0.62	0.13	15M
	256	Shared bank D-NUCA	16.6	0.73	0.12	266M
		SS-energy + shared bank	19.2	0.75	0.12	47M

Table 4.5: Effect of technology models on results

The two designs compared in this subsection are not the only points in the design space. For example, one could view a simply-mapped D-NUCA as an n-level cache (where n is the bank associativity) that does not force inclusion, and in which a line is migrated to the next highest level upon a hit, rather than the highest. A D-NUCA design could be designed to permit limited inclusion, supporting multiple copies within a spread set. Alternatively, a ML-UCA in which the two (or more) levels were each organized as S-NUCA-2 designs, and in which inclusion was not enforced, would start to resemble a D-NUCA organization in which lines could only be mapped to two places.

Cache Design Comparison

Figure 4.3 compares the 16MB/45nm IPC obtained by the best of each major scheme that we evaluated: (1) UCA, (2) aggressively pipelined S-NUCA-1, (3) S-NUCA-2, (4) aggressively pipelined, optimally sized, parallel lookup ML-UCA, (5) DN-best, and (6) an ideal D-NUCA upper bound. This ideal bound is a cache in which references always hit in the closest bank, never incurring any contention, resulting in a constant 3-cycle hit latency, and which includes the PTP search capability for faster miss resolution.

The results show that DN-best is the best cache for all but three of the benchmarks

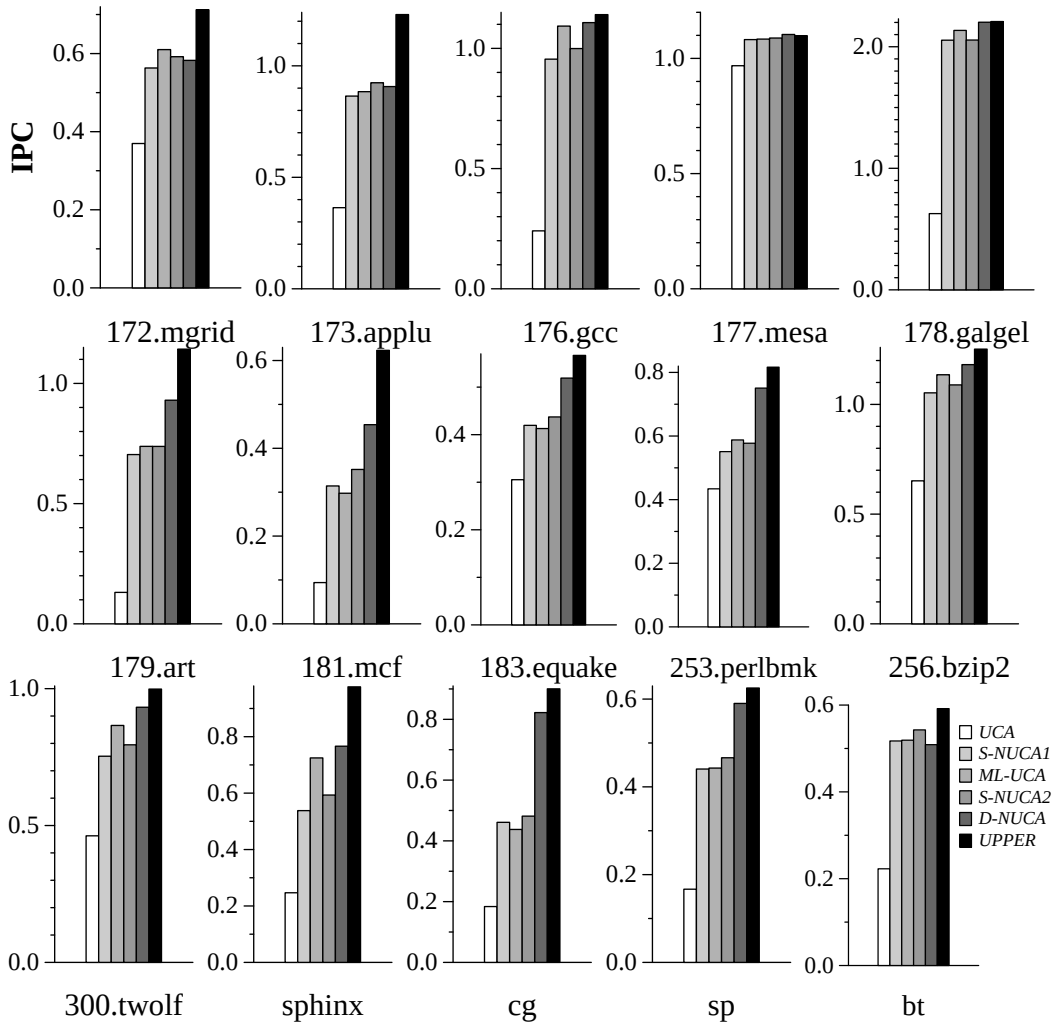


Figure 4.3: 16MB cache performance for various applications including SPEC2000, NAS suite, and Sphinx

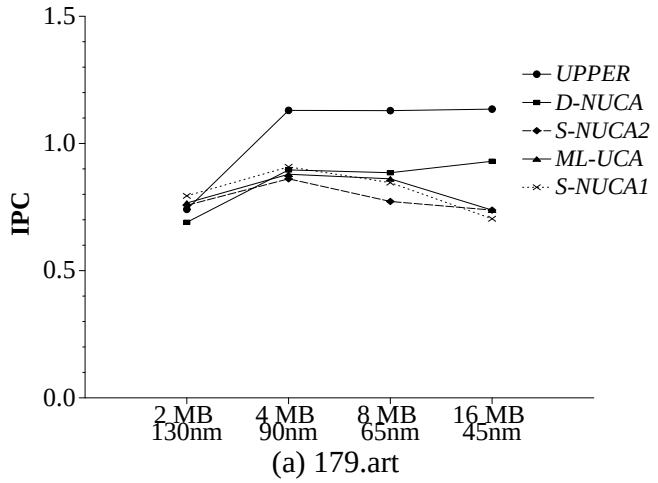
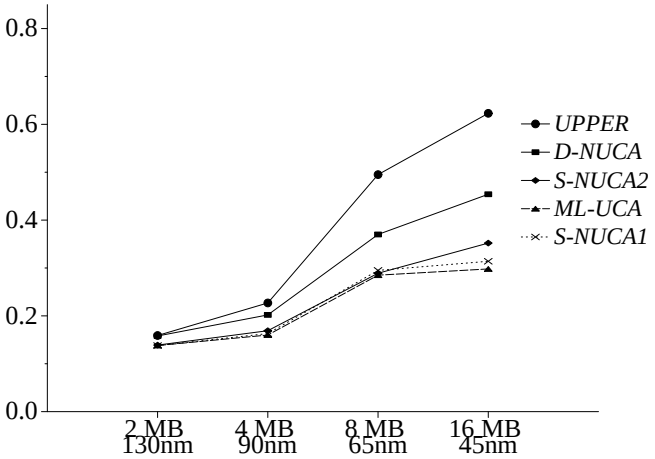


Figure 4.4: Performance summary of major cache organizations : art

(*mgrid*, *gcc*, and and *bt*). In those three, DN-best IPC was only slightly worse than the best organization. The second-best policy varies widely across the benchmarks; it is ML-UCA for some, S-NUCA-1 for others, and S-NUCA-2 for yet others. The DN-best organization thus offers not only the best but the most stable performance. The ideal bound (labeled *Upper* on the graphs) shows the per-benchmark IPC assuming a loaded L2 access latency of 3 cycles, and produces an average ideal IPC across all benchmarks of 0.89. We found that the DN-best IPC is only 16% worse than *Upper* on average, with most of that difference concentrated in four benchmarks (*applu*, *art*, *mcf*, and *sphinx*).

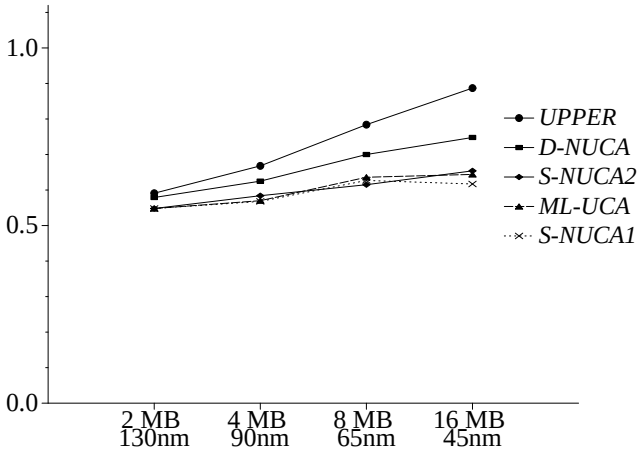
Figure 4.4, 4.5 and 4.6 shows how the various schemes perform across technology generations and thus cache sizes. The IPC of *art*, with its small working set size, is shown in Figure 4.4. Figure 4.5 shows the same information for a benchmark (*mcf*) that has a larger-than-average working set size. Figure 4.6c shows the harmonic mean IPC across all benchmarks.

First, the IPC improvements of D-NUCA over the other organizations grows as the



(b) 181.mcf

Figure 4.5: Performance summary of major cache organizations : mcf



(c) All Benchmarks

Figure 4.6: Performance summary of major cache organizations : AVG

cache grows larger. The adaptive nature of the D-NUCA architecture permits consistently increased IPC with increased capacity, even in the face of longer wire and on-chip communication delays. Second, the D-NUCA organization is stable, in that it makes the largest cache size the best performer for twelve applications, within 1% of the best for two applications, within 5% for one application, and within 10% for one application. Figure 4.4 shows this disparity most clearly in that D-NUCA is the only organization for which *art* showed improved IPC for caches larger than 4MB.

4.2 Chip-Multiprocessor D-NUCA

Chip-Multiprocessors (CMPs) are now commonplace. The major processor companies have adopted CMP designs across various domains; server, desktop and embedded domains. As more transistors are integrated at smaller technologies, more processor cores are expected to be integrated in the chip. While much work exists on building multi-processor systems, the best design for building a scalable CMP is still open research question. In particular, the trend of integrating many cores in a single chip provides a new challenge in designing on-chip memory hierarchy. Even though L1 caches are likely remain private and tightly integrated to processor cores, how to manage L2 caches will be a key design decision to building a scalable CMP.

The L2 caches may be shared by all processors or may be separated into private per-processor partitions. The completely private L2 cache design provides faster access time than the shared design since the private per-processor partition is smaller than the shared cache. In addition, the private L2 design allows a replicated copy of data in individual private partitions, which further reduces the cache access time if a cache hit occurs in a private partition. On the other hand, the completely shared L2 cache design maintains a

single copy of data in the entire shared pool that results in a larger effective cache size and the corresponding lower miss rate.

The tension between a private cache design and a shared cache design is driven by application characteristics. Each application will benefit differently with the reduced hit latencies of a private cache design versus the reduced misses of a shared cache design. The applications with larger working sets and less data sharing benefit more from a shared cache design while the applications with smaller working set and high data sharing get more benefits from a private cache design.

To address the design trade-off between private and shared caches, we first propose a composable cache substrate based on the non-uniform cache architecture (NUCA) design that can be configured as a private cache design or as a shared cache design per-application basis. In addition to the two ends of the spectrum of cache designs (private, shared), the underlying cache substrate permits dynamic selection of any degree of cache sharing, adjusted by the operating system. Here, we define the *sharing degree* as the number of processors that share a given pool of cache. In this terminology, a sharing degree of one means that each processor has its own private L2 partition, whereas a sharing degree of sixteen means that all processor are sharing a single large cache array in a 16-processor CMP system. Since the detailed discussion on the performance effect of the sharing degree was discussed by Huh's dissertation [52], this dissertation focuses on the energy implication of various sharing degree.

Secondly, we evaluate the effect of dynamic data migration in D-NUCA to reduce average L2 hit latencies and thus support larger sharing degrees. While cache designs with a large sharing degree reduce the overall cache miss ratio, the cache hit latencies increase significantly with larger effective cache capacity. In the previous chapter, we showed how

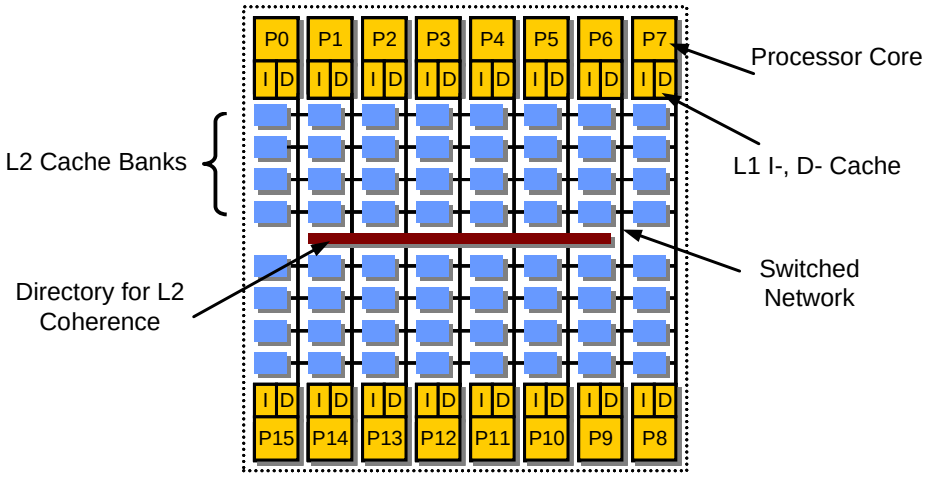


Figure 4.7: Composable cache substrate for flexible sharing degree

dynamic data migration in D-NUCA reduces the average cache hit latencies of a large uniprocessor L2 cache. In the following subsections, we show that dynamic mapping capabilities can potentially reduce long latencies in a large sharing degree CMP cache. We also show that dynamic mapping can reduce the total energy consumed by an on-chip cache subsystem, by reducing the on-chip network traffic in higher sharing degrees.

4.2.1 CMP L2 Cache Design Space

As shown in Figure 4.7, a cache substrate we evaluate to support flexible sharing degree is based on a composable cache substrate that is explored in Chapter 3. The composable cache substrate breaks large on-chip L2 caches into many fine-grained SRAM banks that are independently accessible, with a switched 2-D mesh network [39] embedded in the cache.

The configurable nature of switched network allows caches to be composed to support various sharing degrees. By adjusting the bits used to route memory addresses to a

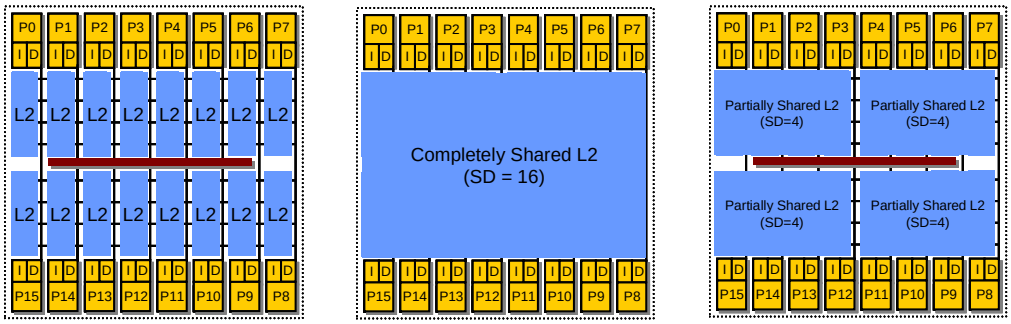


Figure 4.8: Various sharing degrees from the sharing degree one (a), the sharing degree 16 (b), to the sharing degree four (c)

cache bank, the cache array is configurable by the system to use any degree of sharing. If each processor maps the same address bit string to a different bank, the sharing degree is one. If all processors map the same address bits to a single bank, the sharing degree is sixteen.

Lines can be mapped into this array of cache banks with fixed mappings or dynamic mappings, where cache lines can move around within the cache to further reduce the average cache hit latency. With a static mapping policy, a fixed hash function uses the lower bits of a block address to select the correct bank. The L2 access latency is thus proportional to the distance from the issuing L1 cache to the L2 cache bank. By allowing non-uniform hit latencies, static mapping can reduce hit latencies of traditional monolithic cache designs, which fix the latency to the longest path [64]. Because a block can be placed into only one bank, the L2 access latency is essentially determined by a block address.

Parameter	Value
Processor frequency	5 GHz
Issue width	4
Window size	64-entry RUU
Number of CPUs	16
L1 I/D cache	32KB, 2-way, 64B block, 8 MSHRs
L2 cache	8x8 banks
L2 cache bank	256KB, 16-way, 5 cycle latency
Network	1 cycle latency between two adjacent banks
On-chip directory	10 cycle access latency
Main Memory	260 cycle latency, 360 GB/s bandwidth

Table 4.6: Simulated system configuration

Figure 4.8 shows three possible partitioning schemes in a 16-processor CMP that have sharing degrees of one, 16, and four, respectively. With a sharing degree of one (Figure 4.8a), the CMP has sixteen 1 MByte caches, each of which is private to one processor. With a sharing degree of sixteen (Figure 4.8b), the CMP has only one 16 MByte cache, which is shared by all sixteen processors. Figure 4.8c shows the configuration of the sharing degree four,in which four processor cores share 4MB pool of cache banks. In addition to the shown three configurations, the evaluated cache substrate supports the sharing degrees of two and eight as well. To change sharing degrees, the on-chip coherence mechanism must have the flexibility to adapt to different organizations.

Methodology

We evaluated our CMP cache designs using MP-sauce, an execution-driven, full-system simulator [52]. The simulator was derived from IBM's SimOS-PPC, which uses AIX 4.3.1 as the simulated OS. The processor model extends the SimpleScalar processor timing model, adding multiprocessor support. Table 4.6 shows a summary of the main architectural

parameters to measure performance and energy.

The L2 cache bank array is connected with a 2D-mesh point-to-point interconnection network comprised of links and switches. While we model all messages for coherence and data migration to assess network bandwidth, we assume infinite buffering at each switching node. To evaluate the effect of input buffer size on performance, we used a separate cycle-accurate on-chip network simulator with expected network traffic [39]. With the trace-driven network simulation, we confirmed that the increase of input buffer size beyond five entries has little effect on performance compared to the infinite input buffers.

We estimate the dynamic energy consumption of the L2 cache subsystem to investigate the energy consumption effects of varying the sharing degree and using dynamic data migration. We include all L2 cache bank accesses, on-chip directory accesses for coherence management and the partial tag accesses for the D-NUCA design. On a 45nm design at 5GHz, we estimate that the energy consumption ratio of L2 cache bank access: cache line movement per hop: on-chip directory access: partial tag access is about 7:5:2:1. To model the router energy consumption, we use the structural RTL-based energy estimation technique with the Synopsys Primepower tool. The router RTL is obtained from the TRIPS prototype that implemented the S-NUCA L2 cache [99]. We used CACTI [117] to estimate the energy consumption for accessing various SRAM array structures in the L2 subsystem, including cache banks, the on-chip directory and the partial tag structure.

We used three commercial applications: SPECWeb99, TPC-W, and SPECjbb, and four scientific shared-memory benchmarks from the SPLASH-2 suite [126]: Ocean, Barnes, LU, and Radix. Table 4.7 shows the dataset size and other notable features of each application.

Application	Dataset/Parameters
SPECWeb99	Apache web server, file set: 230MB, 480 transactions
TPC-W	185MB databases using Apache & MySQL, 48 transactions
SPECjbb	IBM JVM version 1.1.8, 16 warehouses, 3000 transactions
Ocean	258×258 grid
Barnes	16K particles
LU	512×512, 16×16 blocks
Radix	1M integers

Table 4.7: Application parameters for workloads

4.2.2 Effect of Sharing Degree in CMPs

In this section, we briefly summarize the trade-offs of higher and lower sharing degrees. Then, we discuss the effect of various sharing degrees on the energy consumed in the L2 subsystem.

Hit latency versus hit rate

The main advantage of higher sharing degrees is higher L2 cache hit rates. If the working sets across CPUs are not well balanced, private L2 caches can make one CPU suffer from capacity misses while other CPUs have unused cache space. Shared caches, on the other hand, allow otherwise unused cache space to be used by the space-hungry CPU. Furthermore, shared caches keep at most one copy of a block, not wasting space by storing multiple copies of the same block, unlike private L2 caches sharing copies of the same line. As a result, shared caches can effectively store more data, indirectly increasing hit rates.

However, the drawback of a higher sharing degree is the potential for higher average hit latency due to the larger size, longer wire delays, and increased bandwidth requirements. In future wire-dominated implementations, the effect of increased hit latency may outweigh

the benefit of increased hit rates for shared caches.

On a set of benchmarks (described in Table 4.7), we observed that for shared S-NUCA organizations, low-to-medium sharing, from one to four, provide the best performance for all applications except one. The best sharing degree across all benchmarks is four. We confirmed that significant latency reductions are possible for private L2 caches, and significant miss reductions are possible for shared L2 caches. More detailed evaluations are presented by Huh in his dissertation [52].

Coherence overheads

Inter-processor communication through a shared L2 cache is faster than through private L2 caches connected by a coherent bus. With shared L2 caches, processors communicate through L2 cache blocks directly. As sharing degrees increase, more processor-to-processor communication can be transferred within local shared caches, avoiding slower coherence networks across shared caches. Furthermore, since the size of L1 caches is smaller than the size of L2 caches, modified data in the L1 are frequently flushed to shared L2 caches, making the modified data readily available to other processors in the same shared cache. By absorbing many local communications into shared caches, higher sharing degree caches can reduce slower three-hop cache-to-cache transfers.

Energy efficiency

The sharing degree can affect energy consumed by on-chip network traffic. If the majority of cache accesses hit in small local caches, a lower sharing degree cache can reduce the network traffic. In this situation, most data traffic is localized between processors and close cache banks, reducing traversal distances. However, if data accesses to on-chip remote

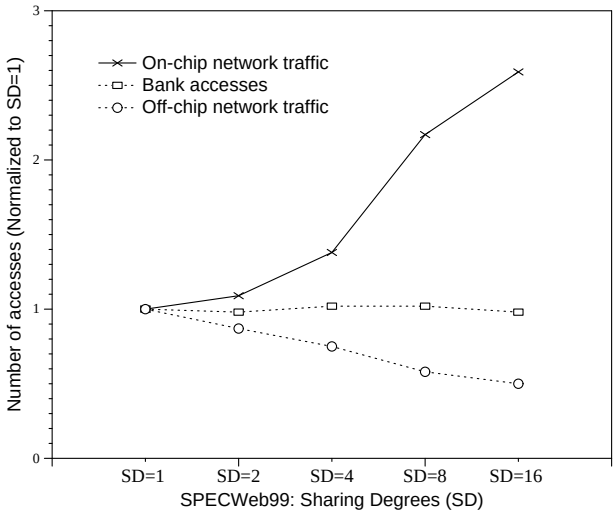


Figure 4.9: On-chip network traffic, bank accesses, and off-chip memory traffic with varying sharing degrees (normalized to SD=1)

caches are frequent, a lower sharing degree may increase network traffic, bank accesses and L2 directory accesses. Furthermore, a higher sharing degree can be more efficient for off-chip memory accesses, since the hit rate can be higher than with a lower sharing degree. If driving off-chip signals and external DRAMs consume a large portion of system power, decreasing off-chip accesses will become critical.

Figure 4.9 presents three energy related statistics: on-chip network traffic, bank accesses and off-chip memory traffic across various sharing degrees. Each statistic is normalized to the sharing degree (SD) of one. Across different applications, these metrics do not change significantly, so we present the result from SPECWeb99.

The most significant change in the energy efficiency is the network traffic increase. The network traffic increases as sharing degrees become higher, since command and data packets must traverse more hops in higher-sharing-degree caches. Between SD=4 and SD=8, the traffic increases sharply, since processors need to access banks on the opposite

of the chip. On-chip network traffic increases by 170% from SD=1 to SD=16. However, up to SD=4, the increase is modest a 35%. A 2-D mesh network consumes less area than a higher degree networks, such as a torus. However, higher degree networks, which can reduce network hop distance at the cost of added area, may be able to reduce hit latencies for higher sharing degrees. Sharing degree changes do not affect bank accesses significantly, but off-chip memory accesses can be affected considerably, depending on the applications. As higher sharing degrees can improve hit rates, off-chip memory traffic decreases.

We draw three conclusions from these results. First, high-degree shared caches for CMPs do not have any advantages in wire-delay dominated future technologies even when high degrees of application sharing exist. The increase in L2 hit latency in shared caches degrades performance more than the reduced misses improve it. Second, the sharing degree can change overall performance significantly. Third, no single sharing degree provides the best performance for all benchmarks. Nevertheless, the SD=4 design point has the best average performance for the applications used in this evaluation, and is the best compromise fixed design point for this mix of workloads on S-NUCA.

4.2.3 Effect of Dynamic Data Migration

Dynamic mapping capabilities can potentially reduce long latencies with large sharing degrees. Performance improvements are achieved when the migration policy is successful and the reduction in latency dominates the increased latency of the more complex lookup mechanism. To isolate the effectiveness of dynamic migration from the overheads of the search mechanism, we evaluated an ideal D-NUCA with a perfect search mechanism (D-NUCA Perfect). The perfect D-NUCA configuration assumes an oracle searching mechanism that allows L1 misses to be sent directly to the L2 bank storing the requested block on a hit. L2

Sharing Degree	SD=1	SD=2	SD=4	SD=8	SD=16
S-NUCA	11.7	12.6	14.3	20.5	24.7
D-NUCA Perfect	8.7	9.2	10.7	15.1	19.1
D-NUCA Real	9.5	10.0	11.4	18.1	21.9

Table 4.8: Average D-NUCA L2 hit latencies with varying sharing degrees

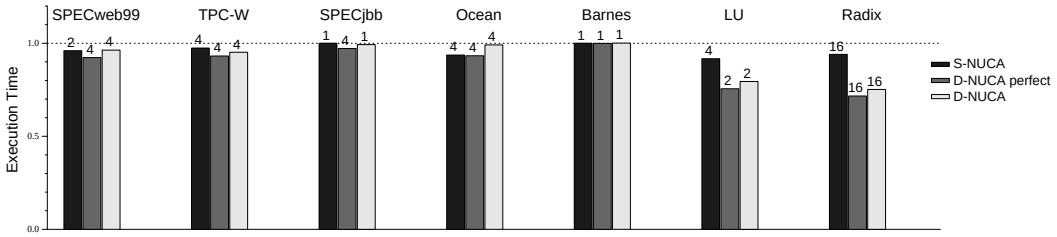


Figure 4.10: D-NUCA execution times (normalized to S-NUCA with SD=1)

misses are also detected without any search overhead in the perfect configuration. However, the perfect D-NUCA configuration still models other overheads such as network and bank bandwidth consumption for accesses and block migration.

Table 4.8 shows the average L2 hit times across all applications for five sharing degrees. With the perfect lookup mechanism, D-NUCA migration policies show significant reductions in L2 hit latencies. The latency reductions increase as the sharing degree increases. At SD=16, the perfect D-NUCA policy reduces the average L2 hit latency by 23% compared to the S-NUCA design. However, with a realistic search mechanism with distributed partial tags, the hit latencies of D-NUCA are significantly increased from the perfect lookup mechanism, confirming that the search mechanism is a key design issue with D-NUCA.

Figure 4.10 shows the relative execution times of the best performing sharing degree

for the S-NUCA and D-NUCA design points across all applications. Each bar shows the SD with the best performance noted at the top. This figure illustrates the following: (1) the performance potential of the perfect search and migration mechanism and how closely the realistic implementations can match them, and (2) performance of the realistic D-NUCA design compared to S-NUCA with the best sharing degree.

The perfect search mechanisms with dynamic migration can reduce execution time by 3-28%, except for Ocean. For Ocean, although D-NUCA reduces average hit latencies, L2 miss rates are increased since blocks are not promoted quickly, and are victimized prematurely by new blocks. For SPECjbb, the performance improvement is small, since SPECjbb does not take advantage of the increased sharing degree, and the effect of dynamic migration is not high at low sharing degrees. With realistic search mechanisms, the performance improvement of D-NUCA can be lost (SPECWeb99 and TPC-W). For LU and Radix, dynamic migration shows large improvements of 21%-25%. LU has a relatively large L1 data miss rate of 12%, but the entire working set nearly fits in the L2 caches. The reduction in L2 hit latencies directly improves performance. In Radix however, external memory accesses dominate performance due to both capacity and conflict misses. Therefore, the best performance for Radix is achieved with a sharing degree of sixteen for both S-NUCA and D-NUCA. Furthermore, the increased bank associativity in D-NUCA reduces conflict misses significantly. D-NUCA enables increased effective associativity since a cache address can be mapped to any cache bank in the same bank set. Since shared caches, especially with high sharing degrees, are prone to conflict misses, the increased associativity in D-NUCA helps avoid certain pathological conflicts.

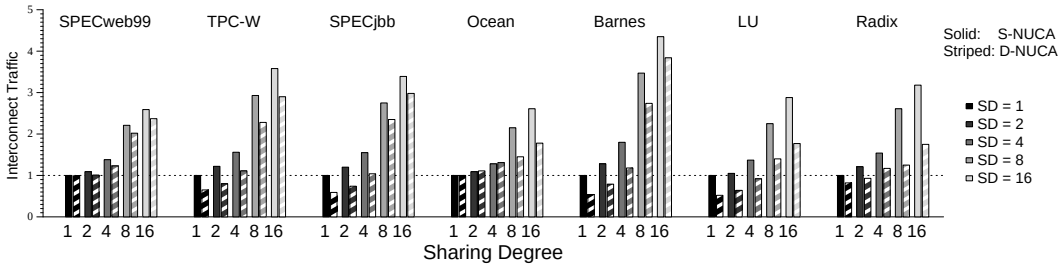


Figure 4.11: On-chip interconnect traffic (normalized to S-NUCA with SD=1)

Results: Energy Trade-Offs

To compare the relative energy consumption of S-NUCA and D-NUCA, we tabulated the power consuming events in the memory system (as in Table 4.6). D-NUCA has the potential to reduce on-chip interconnect traffic by placing frequently accessed blocks close to their requesting cores. However, block migration in D-NUCA generates extra traffic since a migration victim needs to be transferred back to the bank where a hit occurs. D-NUCA also increases the number of bank accesses because three extra bank lookups are necessary for every migration.

Figures 4.11 and 4.12 compare D-NUCA and S-NUCA using two metrics; on-chip interconnect traffic and number of bank accesses. Figure 4.13 presents the total energy consumed by the on-chip L2 cache subsystem. We account for the energy consumed by accessing partial tag arrays in D-NUCA. All numbers are normalized to S-NUCA with the sharing degree of one. S-NUCA numbers are represented by solid bars and D-NUCA numbers are represented by striped bars.

Figure 4.11 shows that placing frequently accessed blocks closer to the processor provides the benefits in reducing the on-chip interconnect traffic. The decreased network

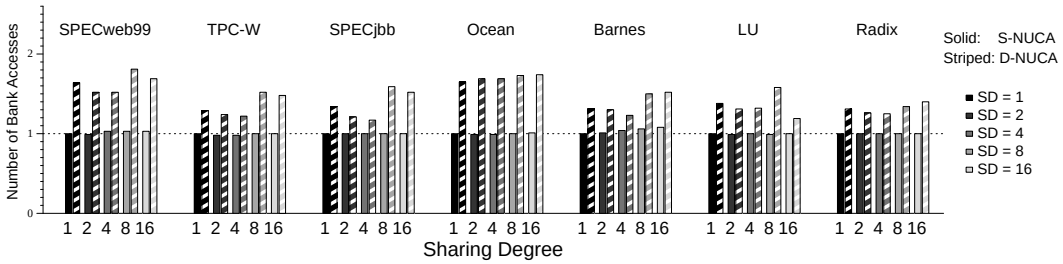


Figure 4.12: Number of banks accesses (normalized to S-NUCA with SD=1)

hops to access blocks are higher than the traffic increase due to migration. D-NUCA with the sharing degree of one effectively reduces the on-chip interconnect traffic by 18% on average compared to S-NUCA. As sharing degree increases, the reduction grows and reaches 45% on average with the sharing degree of sixteen. In terms of network traffic, D-NUCA can be more effective and the gains become higher as sharing degree increases.

As expected, Figure 4.12 shows that block migration in D-NUCA increases the total bank accesses significantly. The number of bank accesses increases by 31-40% for the tested applications with a sharing degree of sixteen, due totally to block migration. Note that the number of bank accesses for D-NUCA increases with sharing degrees of eight and sixteen while the number for S-NUCA remains unchanged across various sharing degrees. This is because of our assumption in floorplanning of processor cores and L2 cache banks. In eight and sixteen sharing degrees, each column bank set is expanded vertically and contains eight cache banks as shown in Figure 4.7. When a block is shared by two processors located in the top and bottom, the block may migrate between eight banks in the column bank set and generate extra bank accesses.

In Figure 4.13, we observe that the total energy consumed by the on-chip L2 cache

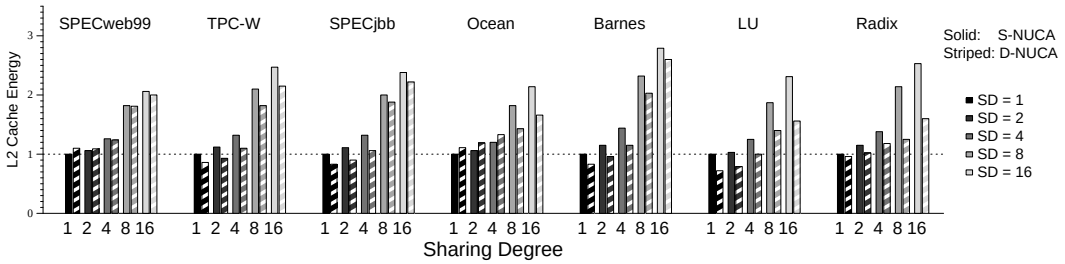


Figure 4.13: Total energy consumed by on-chip L2 cache subsystem (normalized to S-NUCA with SD=1)

subsystem follows the on-chip interconnect traffic trend since the energy consumed by bank accesses and partial tag accesses are relatively small compared to the energy consumed by the on-chip network. Therefore, dynamic migration can contribute to energy reduction as well as performance improvement.

4.3 Summary

Non-uniform accesses are appearing in high performance cache designs [88]. In the first half of this chapter, we evaluate a range of policies to support migrating data dynamically on a composable cache substrate, thereby clustering the working sets within a cache near the processor.

This study shows that uniprocessor D-NUCA cache designs achieve the following four goals:

- *Low latency access*: the best 16MB D-NUCA configuration, simulated with projected 45nm technology parameters, demonstrated an average access time of 17 cycles, which is a lower absolute latency than conventional L2 caches.

- *Technology scalability*: Increasing wire delays will increase access times for traditional, uniform access caches. The D-NUCA design scales much better with technology than conventional caches, since most accesses are serviced by close banks, which can be kept numerous and small with a switched network.

- *Performance stability*: The ability of a D-NUCA to migrate data eliminates the trade-off between larger, slower caches for applications with large working sets and smaller, faster caches for applications that are less memory intensive.

- *Flattening the memory hierarchy*: The D-NUCA design outperforms multi-level caches built in an equivalent area, since the multi-level cache has fixed partitions that are slower than an individual bank. This D-NUCA result augurs a reversal of the trend of deepening memory hierarchies. We foresee future memory hierarchies having two or at most three levels: a fast L1 tightly coupled to the processor, a large on-chip NUCA L2, and perhaps an off-chip L3 that uses a memory device technology other than SRAM.

In the second half of the chapter, we extend the concept of non-uniform cache access architecture to emerging chip-multiprocessors (CMPs) and explore the well-known design trade-off between the lower average hit latency with the private L2 cache design and the larger effective cache capacity with the shared L2 cache design. The CMP L2 cache substrate we evaluate is designed to support both low-latency, private logical caches as well as highly shared caches, simply by adjusting the mapping of the same address on different processors to the L2 cache.

The results show that—compared to private, non-shared L2 partitions—the L2 latency more than doubles for a fully shared cache. The results also show that the fully shared

cache could eliminate a third of off-chip misses. However, the fully shared cache can incur a 170% network traffic increase. Clearly, a large opportunity exists if this gap can be bridged. The S-NUCA organization (static mapping) is best for a low-to-medium sharing degrees for all applications; the extra hit latency is simply too detrimental for larger sharing degrees.

For a subset of applications, we observe that the dynamic data migration capabilities of D-NUCA can reduce the average hit latency, driving the ideal sharing degree higher. In addition, D-NUCA showed the potential benefit of reducing the energy consumption as well by decreasing the on-chip network traffic in higher sharing degrees. However, both performance gains and energy reductions over the S-NUCA design with the best sharing degree are shown to be modest. We conclude that the performance gains of the D-NUCA design are unlikely to justify the added design complexity.

Chapter 5

Composable Processors

Due to limitations on clock frequency scaling, most future computer system performance gains will come from power-efficient exploitation of concurrency. Consequently, the computer industry has migrated toward chip multiprocessors (CMPs), in which the capability of the cores depends on the target market. Some CMPs use a greater number of narrow-issue, in-order cores (Niagara), while others use a smaller number of out-of-order superscalar cores with SMT support (IBM Power5). In the non-server domains, the application software threads must be able to provide sufficient concurrency to utilize all the processors. Another disadvantage of conventional CMPs is their relative inflexibility. In a conventional design, the granularity (i.e., issue width) and number of processors on each chip are fixed at design time, based on the designers' best analyses about the desired workload mix and operating points. Any fixed design point will result in suboptimal operation as the number and type of available threads change over time.

In this chapter, we describe and evaluate a potential alternative, composable processors that build on composable on-chip memories. A composable processor consists of

multiple simple, narrow-issue processor cores that can be aggregated dynamically to form more powerful logical single-threaded processors. Thus, the number and size of the processors can be adjusted on the fly to provide the target that best suits the software needs at any given time. The same software thread can run transparently—without modifications to the binary—on one core, two cores, up to as many as 32 cores in the design that we simulate. Low-level run-time software can decide how to best balance thread throughput (TLP), single-threaded performance (ILP), and energy efficiency. Run-time software may also grow or shrink processors to match the available ILP in a thread to improve performance and power efficiency. Henceforce, we call a composable processor that we evaluate shortly "CLP" (Composable Lightweight Processor).

Figure 5.1 shows a high-level floorplan with three of many possible configurations of a CLP. The small squares on the left of each floorplan represent a single core while the squares labeled L2 on the right represent some form of distributed level 2 cache. The system could obviously decide to run 32 threads on one core each (Figure 5.1a) if the number of available threads were high. If single-threaded performance was paramount, and the thread contained high internal concurrency, the CLP could be configured to run that thread across the number of cores that maximized performance (up to 32, as shown in Figure 5.1c). If energy efficiency was paramount, for example in a data center or in battery-operated mode, the system could configure the CLP to run each thread at its best energy-efficient point, which in our experiments ranges from two to 16 cores per thread, depending on the application. Figure 5.1b shows an energy-optimized CLP configuration running eight threads across a range of processor granularities.

A fully composable processor is signified by three characteristics: (1) serial program execution is distributed over multiple processors, (2) no hardware structures are phys-

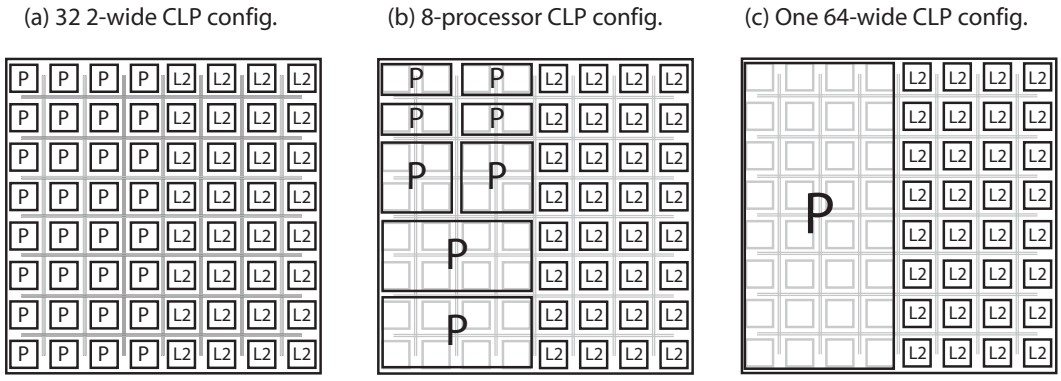


Figure 5.1: Three dynamically assigned CLP configurations

ically shared by these processors, and (3) the number of processors combined to execute a serial program can be dynamically changed transparent to the running application. Creating larger logical microarchitectural structures from smaller ones is the principal challenge for the design of a composable processor. Composing some structures, such as register files and level-one data caches, is straightforward as these structures in each core can be treated as address-interleaved banks of a larger aggregate structure. Changing the mapping to conform to change in the number of composed processors merely requires adjusting the interleaving factor or function.

However, banking or distributing other structures required by a conventional instruction set architecture is difficult. For example, operand bypass (even when distributed) typically requires some form of broadcast, as tracking the ALUs in which producers and consumers are executing is difficult. Similarly, instruction fetch and commit require a single point of synchronization to preserve sequential execution semantics, including features such as a centralized register rename table and load/store queues. While some of these challenges can be solved by brute force, supporting composition of a large number of processing elements can benefit from instruction-set support.

A better-fitting class of ISAs may be explicit data graph execution (EDGE) architectures, which employ block-based program execution and explicit intra-block dataflow semantics, and have been shown to map well to distributed microarchitectures [15]. The particular CLP we evaluate, called TFlex, achieves the composable capability by mapping the large, structured instruction blocks across participating cores differently depending on the number of cores that are running a single thread.

When multiple cores collaborate to run a single thread, all of the distributed resources in each core are used by the thread. Each instruction block is split among the cores, with operands being routed across a scalar operand network [40, 118] to wake up instructions on participating cores. The other resources, such as the L1 instruction and data caches, register files, branch predictors, and load/store queues, each form a physically distributed but logically single resource, using support in the microarchitecture that we describe in Section 5.2.

In this chapter, we describe the TFlex CLP microarchitecture, and compare the performance, area, and power consumed by various configurations against the TRIPS processor, which use the same ISA, as a reference processor. The TFlex CLP microarchitecture allows the dynamic aggregation of any number of cores–up to 32 for each individual thread–to find the best configuration under different operating targets: performance, area efficiency, or energy efficiency. The performance, area, and power models are derived from and validated using the TRIPS hardware. On a set of 26 benchmarks, including both high- and low-ILP codes, results show that the best configurations range from one to 32 dual-issue cores depending on operating targets and applications. The TFlex design achieves a 1.4x performance improvement, 3.4x performance/area improvement, and 2.0x performance^2/Watt improvement over the TRIPS processor. The capabilities offered by CLPs thus permit flexi-

ble execution depending on workload and environmental mixes, making them a good match for future, general-purpose parallel substrates.

5.1 ISA Support for Composability

The TFlex execution model employs an EDGE (Explicit Data Graph Execution) instruction set architecture [15] proposed to better exploit concurrency from applications while handling the growing wire-delay and the power-scaling challenges of modern superscalar processors.

EDGE instruction sets have two distinguishing features. First, they employ *block-atomic execution*, in which groups of instructions execute as a logical atomic unit, either committing all of their output state changes or none (in some sense like transactions). Second, they support *direct-instruction communication* within each block, allowing instructions to specify their dependent instructions in the instruction itself, rather than communicating through a shared namespace like a register file.

In this section, we first describe these two distinguishing features of EDGE ISA in detail. Then, we discuss how these features of EDGE ISA support efficient and flexible composability.

5.1.1 Blocks

An EDGE compiler [107] constructs blocks [75,108] and assigns each instruction to a location within the block. Each block is divided into between two and five 128-byte chunks by the microarchitecture. As shown in Figure 5.2, every block includes a header chunk which encodes up to 32 `read` and up to 32 `write` instructions that access the 128 architectural registers. The read instructions pull values out of the registers and send them to compute

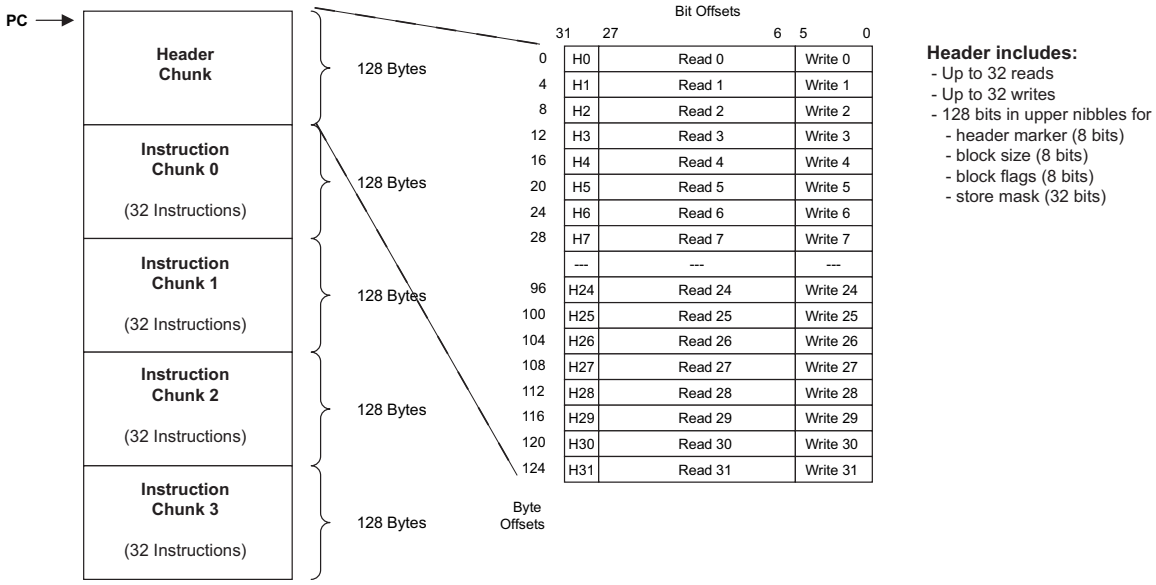


Figure 5.2: Block format (from the paper by Sankaralingam et al. [99])

instructions in the block, whereas the write instructions return outputs from the block to the specified architectural registers.

The header chunk also holds three types of control state for the block: a 32-bit "store mask" that indicates which of the possible 32 memory instructions are stores, block execution flags that indicate the execution mode of the block, and the number of instruction "body" chunks in the block.

A block may contain up to four body chunks–each consisting of 32 instructions–for a maximum of 128 instructions, at most 32 of which can be loads and stores. In addition, all possible executions of a given block must always emit the same number outputs (stores, register writes, and one branch) regardless of the predicated path taken through the block. This constraint is necessary to detect block completion on the distributed substrate. The compiler is responsible for generating blocks that conform to these constraints [107].

5.1.2 Direct Instruction Communications

Direct instruction communication–in which instructions in a block send their operands directly to consumer instructions within the same block in a dataflow fashion–permits distributed execution by eliminating the need for any intervening shared, centralized structures such as an issue window or a register file between the producer and consumer.

As shown in Figure 5.3, the ISA supports direct instruction communication by encoding the consumers of an instruction as targets within the producing instruction, allowing the microarchitecture to determine where the consumer resides and forward a produced operand directly to its target instruction(s). The nine-bit target fields (T0 and T1) shown in the encoding each specify the operand type (left, right, predicate) with two bits and the target instruction with the remaining seven. A microarchitecture supporting this ISA will determine where each of a block's 128 instructions is mapped, thereby determining the distributed flow of operands along the dataflow graph within each block.

More details on the instruction set architecture and execution model are available in the TRIPS ISA Manual [78].

5.1.3 Support for Composability

These features of EDGE ISAs offer power and performance efficiency by removing the overhead of rediscovering dataflow dependences by the hardware since the compiler explicitly encodes the dependences in the ISA. This encoding eliminates the need for most of the unscalable power-hungry structures in the conventional superscalar processors, including associative issue window, complex dynamic scheduler, multi-ported register files, per-instruction register renaming, and complex broadcasting bypass network.

While composability can also be provided using traditional ISAs [54], EDGE ar-

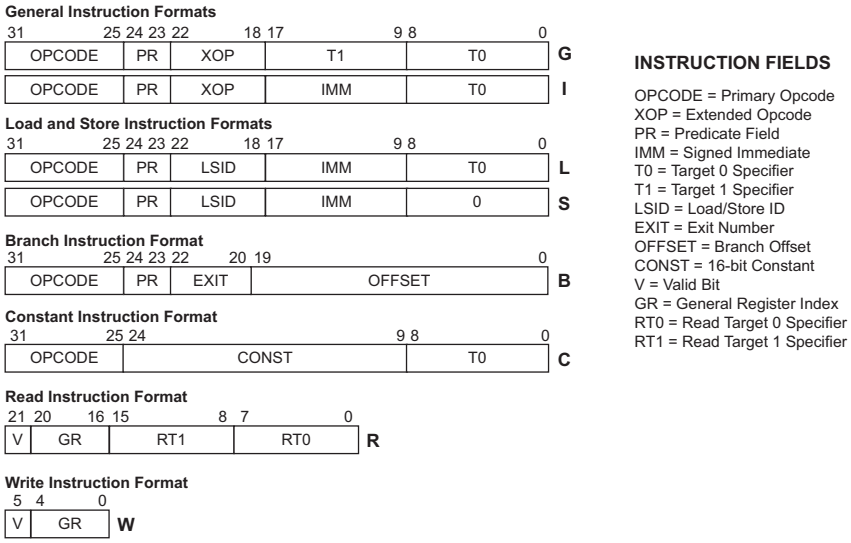


Figure 5.3: Instruction formats (from the paper by Sankaralingam et al. [99])

chitectures provide the salient feature of composability. First, since the dataflow graph is statically and explicitly encoded in the instruction stream, it is simple to shrink or expand the graph on fewer or larger number of execution resources as desired with virtually no additional hardware. Second, when a single thread application runs on multiple cores, traditional architectures will require careful coordination among cores to maintain the sequential semantics of the instruction stream, especially at in-order stages of pipelines such as fetch and commit [54]. This coordination overhead can be significantly reduced if the unit of coordination is done at much larger granularity than individual instructions.

TRIPS is the first architecture to employ an EDGE ISA. It aims to support different granularities of parallelism and takes a partitioning approach, which implements a coarse-grained CMP and logically partitions the large processors to exploit thread-level parallelism when it exists [98]. While this approach is good for improving performance of applications with a moderate number of threads, it provides limited opportunity to adapt to

different performance, power, or throughput needs. In particular, an ultra-large core with SMT support would not be an ideal match for applications that have limited parallelism or applications that have abundant threads. The TFlex architecture takes, conversely, a synthesis approach that uses a fine-grained CMP to exploit thread-level parallelism and tackles irregular, coarser grained parallelism by composing multiple cores into larger logical processors.

5.1.4 ISA Compatibility

Despite the advantages that EDGE ISAs offer, major ISA changes are a daunting challenge for industry, considering the complexity that systems have accumulated. However, several technologies have been developed to transit into a new ISA gracefully under the hood. For example, Transmetra's code morphing software dynamically translates x86 instructions into VLIW code for its processors [23]. We are working on similar techniques and have built a simple PowerPC-to-TRIPS static binary translator. We believe that such static and dynamic translators will enable the easier adoption of new ISAs.

5.2 Microarchitectural Support for Composability

The microarchitectural structures in a composable processor must allow the capacity of the structures to be incrementally added or removed as the number of participating cores increases or decreases. For instance, ideally, doubling the number of cores should double the number of *useful* LSQ entries, double the *useful* storage in branch predictors, etc. To provide efficient operation at a range of composed points, the hardware overheads to support the composability should be kept low. In particular, the hardware resources should not be oversized or undersized to suit either a large processor configuration or a small con-

figuration. At the same time, centralized structures that will limit the scalability of the microarchitecture should be avoided.

To provide this capability, we identify and repeatedly apply two principles. First, the microarchitectural structures are partitioned by address wherever possible. Since addresses of both instructions and data tend to be equally distributed, address partitioning ensures (probabilistically, at least) that the useful capacity increases/decreases monotonically. Second, we avoid physically centralized microarchitectural structures completely. Decentralization allows the size of structures to be grown without the undue complexity traditionally associated with large centralized structures.

This complete partitioning addresses some of the limitations of the original TRIPS microarchitecture. Specifically, the next-block predictor state and the number of data cache banks were limited by the centralization of the predictor and the load-store queue, respectively. Full composability necessitates distributing those structures as well, which provides higher overall performance than the TRIPS microarchitecture irrespective of the composable capabilities. However, those performance gains are a side benefit to the significantly increased flexibility that composition provides.

Figure 5.4 shows how TFlex partitions the microarchitectural structures and interleaves them across participating cores. The microarchitecture uses three distinct hash functions for interleaving across three classes of structures:

- Block starting address: The next-block predictor resources (e.g., BTBs and local history tables) and the block tag structures are partitioned based on the starting virtual address of a particular block, which corresponds to the program counter in a conventional architecture. Predicting control flow and fetching instructions in TFlex occurs at the granularity of a block, rather than individual instructions.

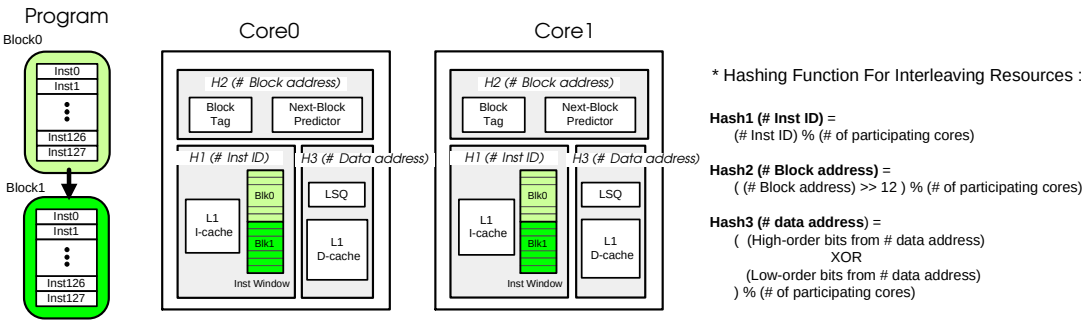


Figure 5.4: An example depicting interleaving of different microarchitectural structures for a two-core processor

- Instruction ID within a block: A block contains up to 128 instructions, which are numbered in order, 0 through 127 as shown in Figure 5.4. Instructions are interleaved across the partitioned instruction windows and instruction caches based on the instruction ID, theoretically permitting up to 128 cores each holding one instruction from each block.

- Data address: The load-store queue (LSQ) and data caches are partitioned by data address from load/store instructions, and registers are interleaved based on the low-order bits of the register number.

In addition, register names are interleaved across the register files. However, because a single core must have 128 registers to support single-block execution, register file capacity goes unused when multiple cores are aggregated. Because interleaving is controlled by bit-level hash functions, the number of cores that can be aggregated to form a logical processor must be a power of 2.

In this section, we first give a brief overview of how the TFlex microarchitecture implements the block-oriented execution model of the EDGE ISA. We describe each of the

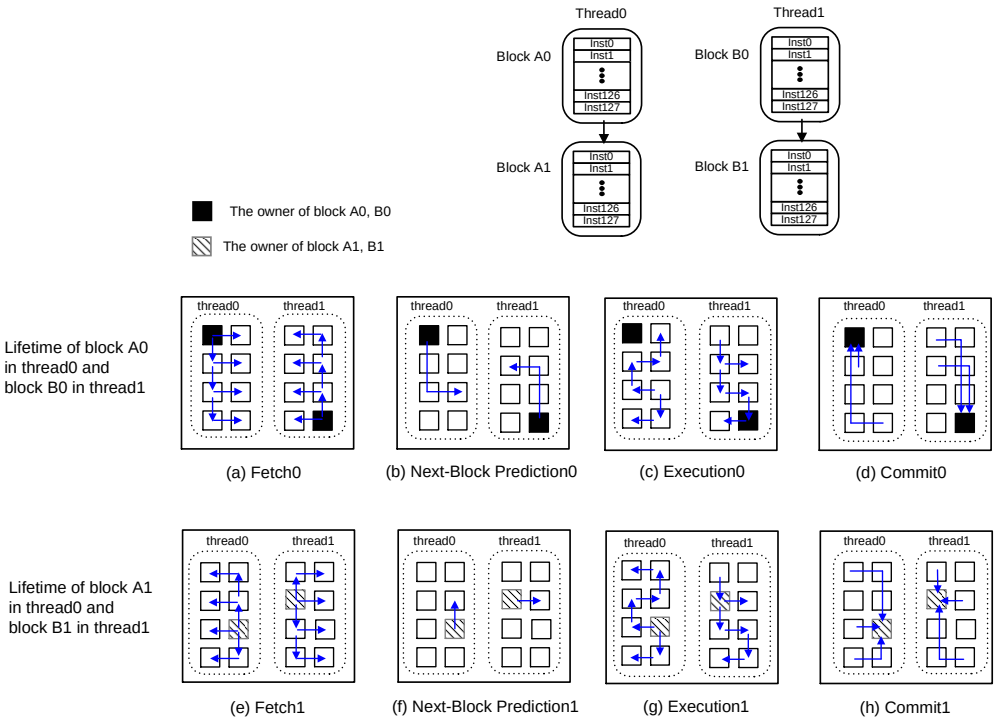


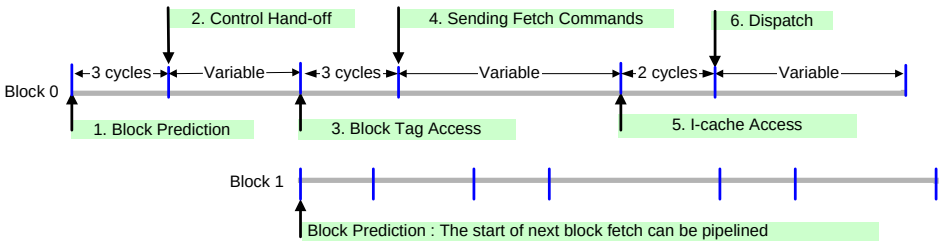
Figure 5.5: TFlex execution stages: execution of two successive blocks (A0, A1) and (B0,B1) from two different threads executing simultaneously on a 16-core TFlex CLP with each thread running on 8 cores

major pipeline stages: block fetch, next-block prediction, block execution, and block commit. Then, we describe the required microarchitectural mechanisms at each stage to support the composable capabilities. we also discuss the challenges in supporting composability in the context of traditional architectures and how the TFlex microarchitecture addresses these challenges.

5.2.1 Overview of TFlex Execution

The basic unit of resource management in the TFlex microarchitecture is a block – a single entry, multiple exit group of instructions – which is fetched and committed atomically. Managing the microarchitectural resources for blocks of instructions rather than individual instructions reduces both the number of resource management operations and the state required for the management. Each in-flight block is assigned an owner core, based on a hash of the block address, which initiates block fetch, predicts the next block, sends the next-block prediction to the core that owns the next predicted block and eventually commits the block. The block core also takes the responsibility of flushing a block when a branch misprediction or load mispeculation is reported.

Figure 5.5 provides an overview of TFlex execution for the lifetime of one block. It shows two threads running on eight cores each. In the block fetch stage, the block owner accesses the I-cache tag for the current block and broadcasts fetch commands to all the participating cores (Figure 5.5a). In parallel, the owner core predicts the next block address and transfers control to the next block owner so that the next block owner can initiate the fetch of the next block (Figure 5.5b). Up to eight blocks may be in flight for eight participating cores. As soon as a fetch command is delivered at each individual core, each core accesses its own I-cache (the instructions in a block are distributed in all participating cores) with address information available from the fetch command, and dispatches fetched instructions into the issue window (Figure 5.5c). When a block completes, the owner detects completion, and when it is notified that it holds the oldest block, it launches the block commit protocol, shown in Figure 5.5d. Figures 5.5e-h show the same four stages of execution for the next block controlled by a different owner; fetch, execution, and commit of the blocks are pipelined and overlapped. Finally, the diagrams show that two distinct programs can be



1. Block Prediction
2. Control Hand-off : Variable depending on the number of participating cores (0 - 10 cycles)
3. Block Tag Access
4. Sending Fetch Commands : Variable depending on the number of participating cores (0 - 10 cycles)
5. I-cache Access
6. Dispatch : Variable depending on the number of cores and the block size (from 1 cycle to 32 cycles)

Figure 5.6: Illustration of different stages of distributed fetch and associated latencies

run on non-overlapping subsets of the cores.

5.2.2 Composable Instruction Fetch

Each core has its own I-cache for storing instructions. With more cores being composed into a larger processor, the overall fetch bandwidth and I-cache capacity scales up. The key challenge in composable instruction fetch is how to maintain the sequential order of the instruction stream among different cores, each capable of fetching independently. Conventional processors will need a centralized unit to coordinate fetches among different cores, especially when the control flow changes in any of cores. This coordination creates the sequential order among fetched instructions.

However, the TFlex execution model eliminates the need to maintain the sequential order through a centralized unit since there is no control flow change within a block (execution is done in dataflow fashion). The sequencing of different blocks is implemented by the following distributed fetch protocol.

Figure 5.6 shows how the TFlex microarchitecture sequences distributed fetch op-

erations, in six stages. After next-block prediction is done, control is transferred to the next block owner using a control hand-off message. The arrival of the hand-off message triggers an I-cache tag access in the new block owner as well as the next-block prediction in parallel. We assume that the I-cache tag for an entire block is maintained by the owner core, while instructions in a block are distributed in each core's I-cache. If there is a hit on the I-cache tag, the owner broadcasts fetch commands that contains the I-cache index and the size of I-cache fetch to all the participating cores. As soon as a fetch command arrives, each core accesses its I-cache and dispatches the fetched instructions into the issue window. Each core can dispatch four instructions per cycle.

The configurable capability of each instruction cache is that the number of instructions from each block that must be mapped to each slave I-cache bank changes depending on the configuration. In 32-core mode, only four instructions from a block are mapped to each node. This fine-grain distribution requires more expensive L1 I-cache misses, as blocks must be fetched from the L2 and distributed to all the participating cores. However, the tag overhead does not increase, since the tags are associated with the blocks, not each individual instruction.

5.2.3 Composable Control-flow Prediction

Control-flow prediction structures are one of the most challenging of all structures to partition for composability. The key challenge is how to distribute state that has been traditionally handled in a logically centralized manner. For instance, when a branch resolves in a core that is different from an owner core for a block, how, when, and where should the predictor be trained and repaired? Further, rate of prediction needs to match or exceed the fetch rate leading to using very minimal communication between different predictors

during time-critical operations.

Similar to the TRIPS prototype microarchitecture, the TFlex control flow predictor issues one next-block prediction for each 128-instruction hyperblock—a predicated single entry, multiple exit region—instead of one per basic block. Predication of hard-to-predict branches within a single block can potentially increase the prediction accuracy. The main difference from the TRIPS predictor is that the TFlex composable predictor treats the distributed predictors in each composed core as a single logical predictor, exploiting the block-atomic nature of the TRIPS ISA to make this distributed approach tenable. The owner core for a block is responsible for generating the prediction for the successive block, and sending that prediction to the next owner core.

The TFlex next-block predictor uses an Alpha 21264-like local-global tournament exit predictor and a target predictor comprising a branch target buffer, a call target buffer, a return address stack and a branch type predictor. To perform distributed block exit and target prediction, several extensions are necessary. All communication to maintain the predictor resources is carefully designed to be done with point-to-point messages.

The local history table naturally supports address partitioning since the next block prediction is performed by the block owner core and the block owner is determined by hashing on the block address. The block prediction that maps to a given core will always map to that core, preserving local histories. To support global prediction, the global history register is transmitted from core to core as each prediction is made. Since the prediction tables in each core are small, each predictor uses history folding (splitting and XORing parts of longer histories) to support longer histories and reduce destructive aliasing. On a flush, the core owning the block signals misprediction, initiates the correct fetch and re-sends the rolled-back global history vector to the new block owner.

For target prediction, the type predictor and the branch and call target buffers are address partitioned. The return address stack is sequentially partitioned across all the cores. The stacks from all the participating cores form a logical global stack. Calls and returns send messages to update the stack top in the appropriate core. In addition, they also send the updated top of the stack value to the next block owner core (just as the global histories are sent). This communication avoids additional penalty of fetching the top of the stack from a different core in case the next block fetched has a return branch. We present the detailed analysis of the predictor in Section 5.3.2.

5.2.4 Composable Instruction Execution

The twin goals of CLP microarchitectural mechanisms for instruction execution are (1) tracking the data dependence information across different cores, and (2) trying to keep dependent instructions as close to each other as possible.

To support execution across a variable number of cores in the conventional superscalar processors, the data dependence information must be identified and stored together when an instruction is steered and slotted into the individual core. While the dependences between pairs of instructions within a core can be tracked by a distributed local register alias table at each core, a centralized global register rename table is required to resolve dependences across different cores. With more cores participating, the number of ports required to sustain the total rename bandwidth becomes prohibitive. The instructions in an EDGE ISA contain the dependence information, eliminating the power-hungry, centralized register rename table.

The TFlex architecture couples compiler-driven assignment of instructions numbers with hardware-determined issue order to minimize communication delays statically

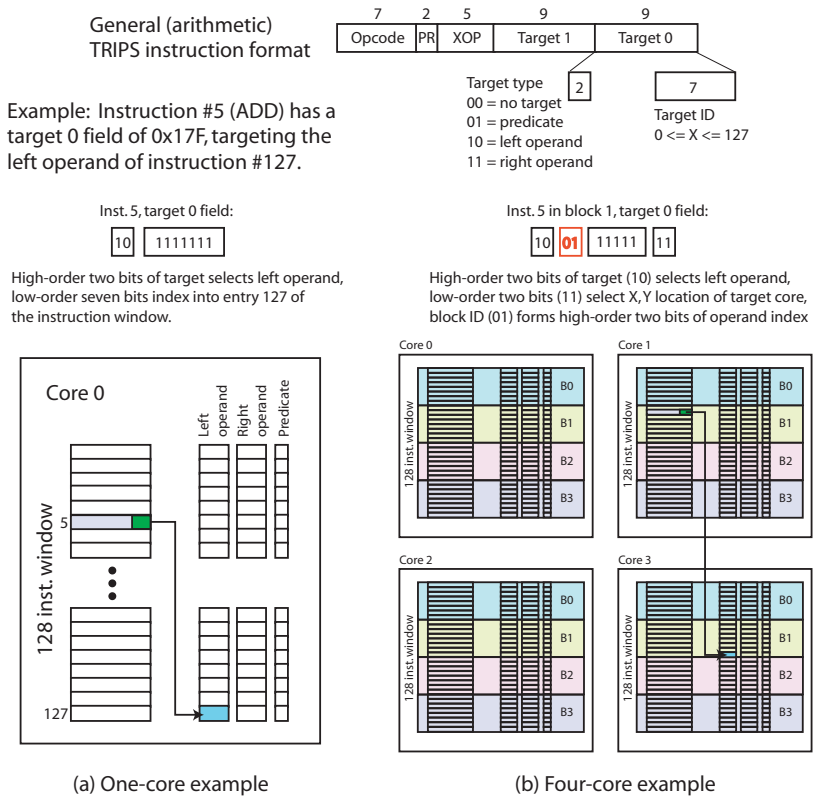


Figure 5.7: Block mapping for one-core and four-core processors

and tolerate uncertain latencies dynamically. The static mapping of instructions to execution resources, in particular, makes the TFlex architecture amenable to distributed and composable substrates. Each core only needs to reinterpret the static mapping between an instruction and its physical location depending on how each core is composed.

Figure 5.7 shows the mechanism that TFlex uses to permit issue across a variable number of composed cores. Each instruction in a TRIPS block contains at least one nine-bit *target* field, which specifies the location of the dependent instruction that will consume the produced operand. Two of the nine bits specify which operand of the destination instruction is targeted, and the other seven bits specify which of the 128 instructions is targeted.

Figure 5.7a shows how the target bits are interpreted in one-core mode, if instruction five is targeting the left operand of instruction 127. All seven bits are used to index into the single instruction block held in the 128 instruction buffers.

Figure 5.7b shows how the microarchitecture interprets the target bits when running in a four-core configuration. The four cores can hold a total of four instruction blocks, but each block is striped across the four participating cores (thus, each core hold 32 instructions from each of the four blocks in-flight). In this configuration, the microarchitecture uses the low-order two bits from the target to determine which core is holding the target instruction, and the remaining five bits to select one of the 32 instructions on that core. When instruction five issues, the microarchitecture uses the low-order two bits to route the operand to the correct core, using the dynamic block identifier and the five remaining target bits to index into the instruction window and wake up the destination instruction.

A key question is how much latency is incurred communicating from core to core, and how much performance is lost as a result. In the TFlex design, the cores are connected by a two-dimensional mesh network. Figure 5.8 shows the datapath from the output of an ALU in one core to the input of an ALU in an adjacent core and illustrates cycle-by-cycle activities when the execution result at core 0 is bypassed into core 1. While dependent instructions can issue back-to-back within one core, there is a one-cycle bubble between two dependent instructions for each network hop an operand must travel. Only a one-cycle pipeline bubble is required for adjacent cores because the operand network sends a control packet a cycle in advance of the data, permitting wakeup to happen in advance of the operand arrival. Area estimates for 65nm indicate a core-center to core-center distance of 1.5mm, corresponding to an optimally repeated wire delay of 170ps. With a fast router that matches the wire delay, the total path delay would be less than 350ps and a one-cycle

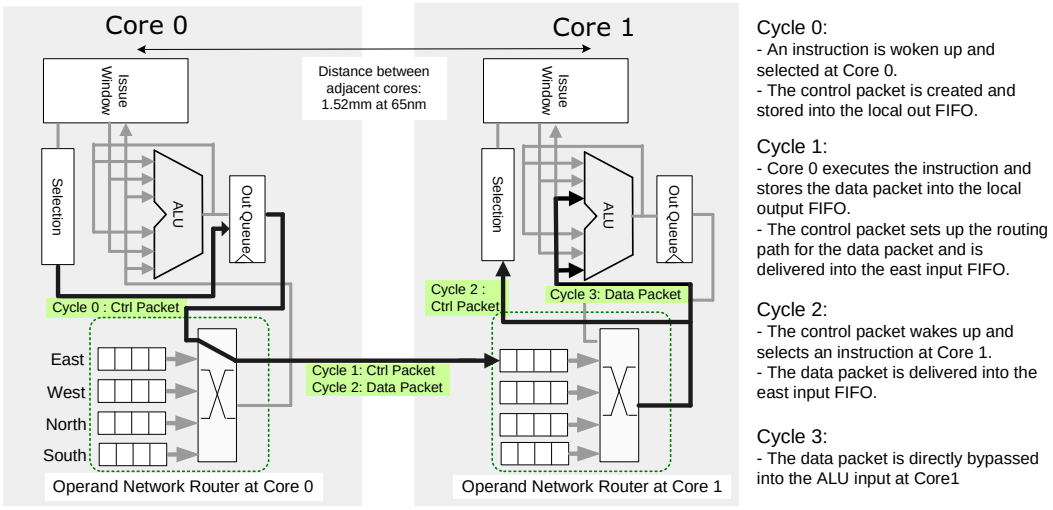


Figure 5.8: Inter-core operand communication

inter-core hop latency could be supported at well over 2.5GHz. Section 5.3.3 contains the detailed operand network analysis with various hop latency and bandwidth assumptions.

5.2.5 Composable Memory System

As with clustered architectures [69, 94], L1 data caches in a CLP can be address partitioned and distributed into each core. When running in single-core mode, each thread can access only its own bank. When multiple cores are composed, the L1 cache becomes a cache-line interleaved aggregate of all the participating L1 caches. With each additional core, each running thread obtains proportionally greater L1 D-cache capacity and an additional memory port. The cache bank accessed by a memory instruction is determined by XORing the high and low portions of the virtual address modulo the number of participating cores. All addresses within a cache line will always map to the same bank in a given configuration. However, unlike conventional architectures, when a core computes the effective address of

a load, the address and the target(s) of the load are routed to the appropriate cache bank, the look-up is performed, and the result is directly forwarded to the core containing the target instruction.

One of the microarchitectural challenges to support a composable memory system is the efficient handling memory disambiguation on a substrate with a variable number of cores. Each TFlex core relies on an unordered, late-binding load store queue (LSQ) structure [103] to disambiguate memory accesses dynamically. As more cores are aggregated to construct a larger window, more entries in the LSQ are required to track all in-flight memory instructions. Partitioning LSQ banks by address and interleaving them with the same hashing function as the data caches is a natural way to build a large distributed LSQ. However, unless each LSQ bank is maximally sized for the worst case (the instruction window size), the system should be able to handle the situation when a particular LSQ bank is full, and thus cannot slot an incoming memory request. (called ”LSQ overflow”). Prior work has shown that both throttling fetch to prevent LSQ overflow and flushing on overflows cause significant performance losses [102]

The TFlex microarchitecture uses a low-overhead mechanism that exploits the functionality of the underlying scalar operand network to make flushes extremely rare [103]. The microarchitecture reserves a fraction of each LSQ (4 entries) for the non-speculative block in flight. If an LSQ bank is full, and a load or store from the non-speculative block arrives, the pipeline is flushed and the non-speculative block is run in single-block mode to guarantee forward progress. If a load or store from one of the speculative blocks arrives at a core where its LSQ bank is full, the request is sent back to the issue window with a negative-acknowledgement (NACK) message and waits until the LSQ bank has an available slot. The question of when to re-issue a NACKed memory instruction imposes an important

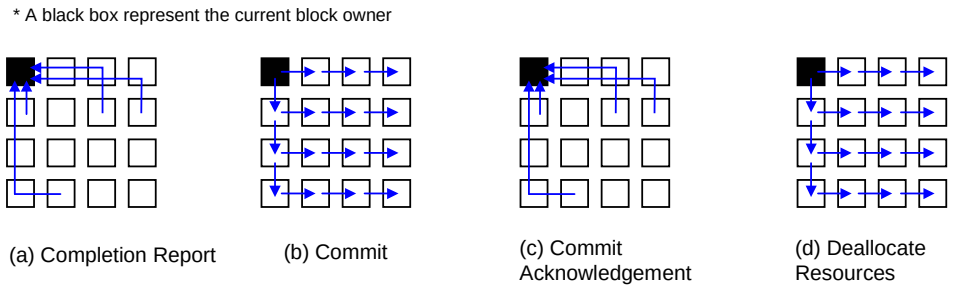


Figure 5.9: Four-stage commit procedure in TFlex

trade-off between the amount of ILP and operand network congestion. We examine a range of policies to determine the optimal configuration in Section 5.3.4.

5.2.6 Composable Instruction Commit

To sequence the committed instruction stream among different cores, traditional architectures must be able to coordinate multiple cores to retire instructions in lockstep. With more cores participating, the overhead of exchanging signals to support lockstep commit increases.

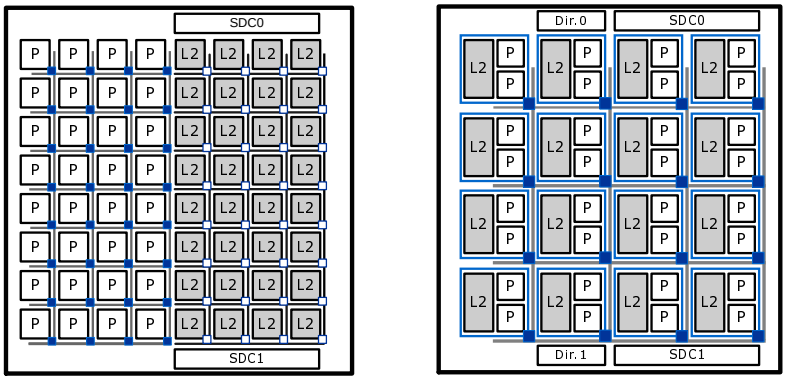
The TFlex architecture reduces the overheads of coordination across different cores by committing a group of instructions en masse. To commit a block, the following four-stage protocol is used, adding one extra stage compared to the three-stage protocol in TRIPS. First, the block owner detects that a block is complete because the block has emitted all of its outputs, consisting of stores, register writes, and one branch (Figure 5.9a). The second stage occurs when the block in question is the oldest block, at which point the block owner sends out a *commit* command (Figure 5.9b). All distributed cores write their outputs to architectural state, and when finished respond with *commit acknowledgement*

signals (Figure 5.9c). Finally, the block owner broadcasts the resource deallocation signals, at which point, the youngest block owner can initiate its own fetch and overwrite the committed block with a new block (Figure 5.9 d). This final stage, which is not present in the three-stage commit protocol in the TRIPS architecture, is required in the TFlex architecture. While TRIPS has a single centralized block owner, each participating core in TFlex can be a block owner based on the block address. If the youngest block owner and the oldest block owner differ in TFlex, the youngest owner must be informed from the oldest block owner that it is safe to initiate a new block fetch and overwrite the oldest block with a new block. Note that the commit steps from Figure 5.9a to Figure 5.9d can be pipelined across different blocks, thereby reducing the effect of handshaking overhead on overall performance. In Section 5.3.1, we measure the overhead of commit coordination and its effect on performance.

An alternative way to initiate the fetch of a new block is to rely on point-to-point communication from the oldest owner to the youngest owner informing it of resource deallocation. In order to enable this, the oldest owner must know the location of the youngest owner, which is discovered earlier when the new youngest owner is identified and a control message is sent to the oldest owner. Point-to-point messages reduce the total message traffic and consume less power with a more complex communication protocol than a simple broadcast protocol. The design trade-off between the amount of message traffic and the complextity of the communication protocol is an interesting open question.

5.2.7 Level-2 Cache Organization for Composable Processors

We explore two design choices to organize L2 caches in CLPs. The first choice is the decoupled L2 organization (Figure 5.10 a) that separates the processor core regions from



(a) Decoupled L2 Organization (b) Integrated L2 Organization

Figure 5.10: Different L2 organizations

the L2 bank regions. With the decoupled L2 design, distance between cores in a composed processor becomes shorter than the integrated L2 design, which provides significant benefits to applications that are more sensitive to operand delivery latency.

The integrated L2 organization (Figure 5.10 b) combines processor cores and L2 cache banks into homogeneous building blocks. The integrated L2 organization can localize communication between cores and L2 banks within a building block, thus removing data transport delay to and from the corresponding L2 bank. Applications that require high bandwidth channels between cores and L2 banks will favor this design.

In addition to latency and bandwidth differences, the decoupled L2 organization makes it easy to expand/shrink L2 cache capacity with relatively small design changes. Many processor vendors offer products with various L2 capacities depending on target market segments and fabrication process maturity. Changing the L2 capacity in the integrated L2 design has more design constraints, since the increased size of a building block can affect the latency between hops to deliver operands in a composed processor.

Finally, while more tightly packed processor cores provides latency benefits in the decoupled design, the integrated L2 design has a better physical design for avoiding hot spots by spreading the cores across in the entire chip. In the integrated design, the processor cores are spread over the entire chip and each "hot" core is surrounded by "cool" L2 cache banks.

Private L2 design Versus. shared L2 design: Another important design trade-off in L2 caches is whether to manage L2 as private caches or as shared caches. In Section 4.2, we described the detailed trade-off between the private L2 design and the shared L2 design. To summarize briefly, private L2 caches provide shorter latency at the expense of lower effective on-chip cache capacity.

While both the decoupled and the integrated L2 organizations do not restrict themselves to either the private or shared design, we choose the shared design for the decoupled L2 organization and the private design for the integrated L2 organization based on our simulation results. Especially, with the integrated organization, the private cache design allows L2 caches to be interleaved with the same hashing function as the L1 caches, which eliminates the need to route a L1 fill request to a L2 cache bank in a different building block.

We evaluated directory protocols to maintain coherence for both private and shared L2 designs: coherence among multiple L1 caches in the shared L2 design and coherence among multiple L2 caches in the private L2 design.

For the coherence of L1 caches in the shared L2 design, the tag of an L2 cache line contains the sharing status vectors to indicate which L1 caches have copies of the line. In the private L2 design, we use a centralized L2 tag directory (but physically partitioned into two, each one is located next to the SDRAM controllers as shown in Figure 5.10). When an L2 miss is detected, the request is sent to the centralized L2 tag directory, which decides

whether to obtain data from another L2 cache on the chip or whether to issue an off-chip memory request.

Managing both L1 and L2 coherence is designed to be oblivious to how each processor is composed. For example, in the shared L2 design, the sharing status vector in the tag keeps track of L1 coherence by handling each L1 cache as an independent coherence unit, which requires enough bits in the status vector for representing all L1 caches. This configuration-independent coherence management allows us to avoid L1 cache flushing on reconfiguration (described in Section 5.2.8). When the new mapping results in L1 misses, the underlying coherence engine can correctly forward the request to L1 caches in the old mapping (if necessary).

5.2.8 Microarchitectural Reconfiguration

There are many factors affecting the ideal number of cores allocated to a single composed processor. One set of factors is the desired metric: performance, throughput (performance/area), or energy efficiency. Many factors besides issue width affect performance. When cores are added to a logical processor, they provide a linear scaling of resources, such as memory ports, issue window capacity and register file bandwidth. When running a thread in the largest, 32-core, 64-wide issue configuration, that processor has a 4K issue window, a 1.2K entry LSQ, a 256-Kbit next-block predictor, and 256KB L1 instruction and data caches with 32 independent banks.

To adjust the configuration, the running processes on the hardware to be adjusted need to be interrupted. The instruction caches must be invalidated, since the instruction mapping across cache banks will change. The registers must be saved and copied into the new configuration according to the new interleaving degree. Finally, two control registers

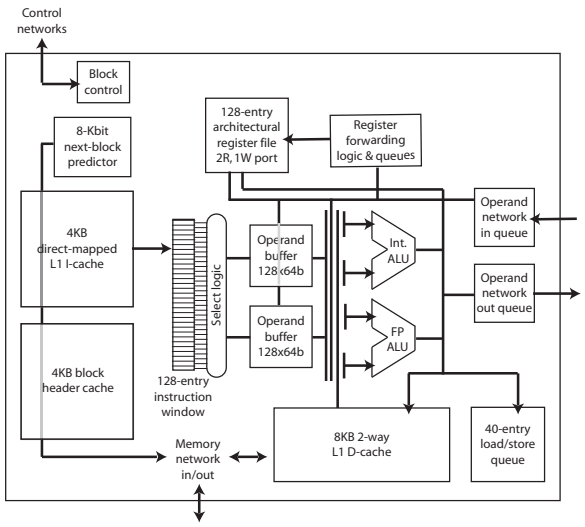


Figure 5.11: Single core TFlex microarchitecture

need to be written in each participating core, specifying the size of the logical configuration and the ID of each core within that configuration. To support composing cores across non-contiguous processor cores, each core needs a mapping table between the ID of each core and its physical location (coordinates) on a processor substrate. At that point, the logical processor(s) can be restarted. The data caches do not need to be flushed, since the new mapping will result in misses, which will be handled correctly by the cache coherence logic.

5.3 Microarchitecture Evaluation

In this section we evaluate the TFlex microarchitecture, with particular emphasis on measuring the overheads of distributed execution with respect to fetch, commit, control-flow prediction, operand delivery and memory disambiguation. Figure 5.11 shows the microarchitecture of a single TFlex core and Table 5.1 summarizes the microarchitectural param-

Parameter	Configuration
Instruction Supply	Partitioned 8KB I-cache (1-cycle hit), Local/Gshare Tournament predictor (8K bits, 3-cycle latency) with speculative updates; Local: 512(L1) + 1024(L2), Global: 4096, Choice: 4096, RAS: 128, BTB: 2048.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP). 128 architectural registers
Data Supply	Partitioned 44-entry LSQ bank, Partitioned 8KB D-cache (2-cycle hit, 2-way assoc, 1-read port and 1-write port). 4MB S-NUCA L2 cache [64] (8-way assoc, LRU, the L2 hit latencies vary from 5 cycles to 27 cycles depending on memory addresses) Average (unloaded) main memory. latency is 150 cycles
Interconnection Network	Each router uses round-robin arbitration. There are four buffers in each direction per router. The hop latency is 1 cycle.

Table 5.1: Microarchitectural parameters for a single TFlex core

eters. The size of the structures ensure that one block can atomically execute and commit. For instance, the instruction window can hold all 128 instructions in a block and the LSQ must be large enough to hold at least 32 load/store instructions, etc.

Table 5.2 summarizes the simulator and the benchmarks we use for the study. To model the TFlex microarchitecture, we wrote an execution-driven simulator. When configured to have the same number of resources as the TRIPS prototype processor [99], this simulator reports performance within 7% of real system measurement on a set of microbenchmarks. We use two different benchmark suites: a hand-optimized suite and a compiler-generated suite. The hand-optimized suite consists of seven EEMBC 2.0 benchmarks, two from Versabench [93], and three signal processing kernels from MIT Lincoln Labs. The compiler-generated suite consists of eight integer and six floating point SPEC benchmarks that are currently supported in the infrastructure.

For the TFlex configurations, these programs are scheduled by using the TFlex instruction scheduler, which differs from the TRIPS scheduler [21] in the following ways.

Simulator		Execution-driven simulator validated to be within 7% of real system measurement.
Benchmarks	EEMBC	a2time01, autocor00, basefp01, bezier02, dither01, rspeed01, tblook01
	LL Kernel	corner turn (ct), convolution (conv), genetic algorithm (genalg)
	Versabench	802.11b, 8b10b
	SPEC INT	164.gzip, 176.gcc, 186.crafty, 197.parser, 253.perlbmk, 255.vortex, 256.bzip2, 300.twolf
	SPEC FP	168.wupwise, 171.swim, 172.mgrid, 173.applu, 200.sixtrack, 301.apsi

Table 5.2: Simulator and Benchmarks

First, the scheduler assumes the 32-core configuration for scheduling instructions. We found that performing instruction scheduling for a larger number of cores and running it on fewer cores results in little performance degradation. Second, the TFlex scheduler considers the differences between TFlex and TRIPS in terms of their distribution of registers and L1 data cache banks. TFlex distributed 128 registers among all participating cores while TRIPS maintains registers only in the top four register tiles. Likewise, TFlex distributes L1 data cache banks into all participating cores, while TRIPS has all of the L1 data cache banks in the left column of tiles. The TFlex scheduler thus reasons about register placement but eliminates the memory placement heuristics in the TRIPS scheduler.

5.3.1 Distributed Fetch and Commit Overheads

Distributed Fetch: Figure 5.12 shows the breakdown of average latencies for the components of the distributed fetch protocol shown in Figure 5.6. Three components of the six components of the fetch mechanism, block tag access, block prediction and I-cache access incur a constant total latency of seven cycles for a block (except for the 1-core configuration, in which there is no next-block speculation and hence the prediction latency is zero).

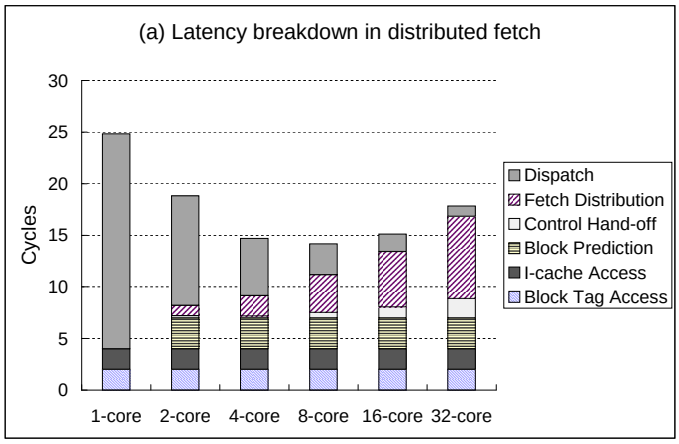


Figure 5.12: Distributed fetch overheads

Of the remaining three components, control hand-off and the fetch command distribution are communication latencies due to distributed execution. The last component, the dispatch latency, which is the latency to fetch from I-cache into the instruction window, incurs a variable latency depending on the number of instructions dispatched at each core.

Figure 5.12 shows that the overall fetch latency depends on the number of cores and is a balance between the variable overheads of control hand-off, fetch distribution, and dispatch. The largest increase comes from broadcasting the fetch command over the multi-hop network to all participating cores, which dominates when 16 or more cores are aggregated. Conversely, the effective dispatch bandwidth increases linearly with the number of cores, and the time to dispatch becomes a very small fraction of the overall latency at 16 or more cores.

Distributed Commit: As described in Section 5.2.6, the distributed commit protocol in TFlex consists of four stages: (1) send commit signals to all cores, (2) update architectural state including store and register file commit, and (3) send “commit complete”

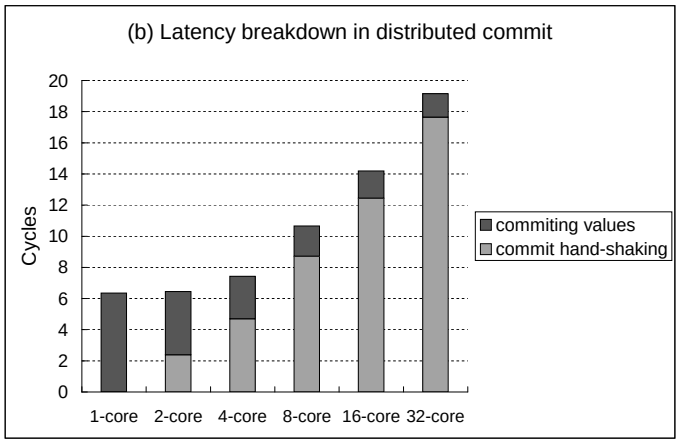


Figure 5.13: Distributed commit overheads

signals back to the block owner (4) send signals to all participating cores for deallocating the hardware resources. Figure 5.13 shows the latency of the two principal components of commit: updating architectural state and handshaking across multiple cores (including sending commit signals, sending "commit complete" signals, and sending "resource deallocation" signals). As expected, the handshaking overhead increases with the number of cores while the architectural state update latency decreases because the register file and data cache bandwidth increase linearly with the number of cores.

Summary: While these latencies can be significant, they will not affect performance if they are not on the critical path. To quantify the performance impact of the coordination overheads of fetch and commit, we simulated an architecture in which all of the distributed handshaking occurs instantaneously. We observed that the performance degradation was less than 2% for the largest composition (32 cores), indicating that the overheads of distributed fetch and commit can be amortized by a block-structured ISA.

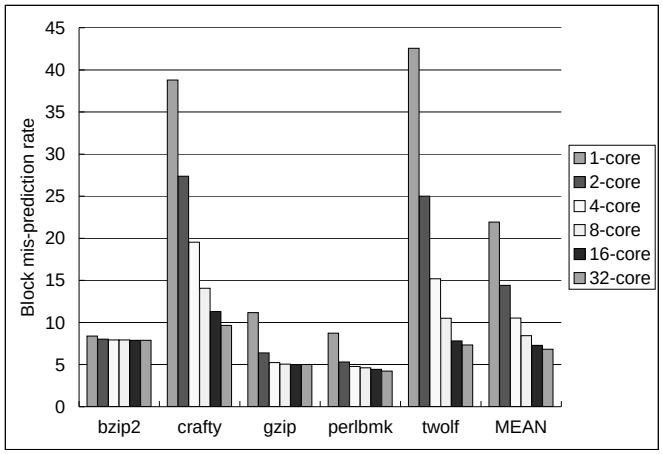


Figure 5.14: Distributed next-block predictor misprediction rates from 1-core to 32-core configuration

5.3.2 Distributed Block Prediction Overheads

Figure 5.14 (a) shows the misprediction rate for five of the SPEC Integer benchmarks across various numbers of cores. As described in Section 5.2.3, the TFlex block predictor address-partitions the local predictor resources among participating cores while transmitting the global information with point-to-point communication. In Figure 5.14, the overall misprediction rate decreases from 19.94% to 7.28% as more cores are aggregated. Though small core configurations show high miss rate, the associated misprediction penalty is low since the block speculation depth is proportional to the number of participating cores. The average miss rate of 7.28% in the 32-core configuration seems high, but we observed that the MPKI (Mispredictions Per Kilo Instructions) number is 2.9 for the 32-core configuration and the number is comparable to some of the best conventional branch predictor like the PAs/Gshare hybrid predictor.

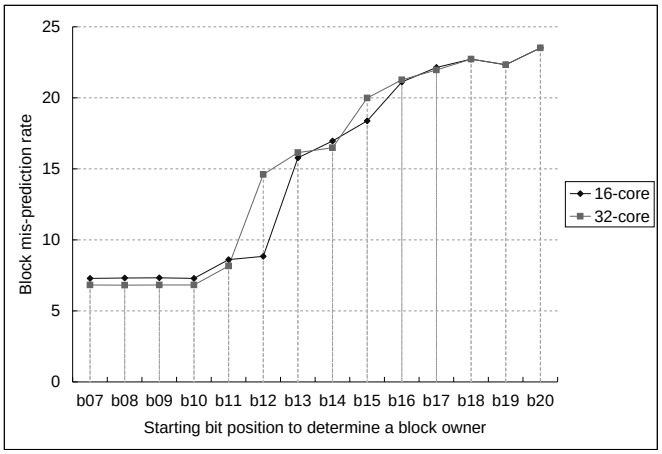


Figure 5.15: Average misprediction rate for 16-core and 32-core with various starting bit positions to determine a block owner

The low MPKI combined with the fact that a block can contain multiple control flows and potentially hide the hard-to-predict branches can be used to explain the higher performance of the TFlex microarchitecture compared to the TRIPS which also uses blocks but does not have the ability to aggregate branch prediction state. Although the small core configurations have an undesirable high MPKI, the performance penalty is not high because they intrinsically run at lower performance levels.

Unlike conventional monolithic branch predictors, a distributed branch predictor makes an interesting trade-off between the prediction accuracy (Figure 5.15) and the communication overheads (Figure 5.16) to reach the distributed predictor resources.

In the TFlex block predictor, a block owner takes charge of predicting the next block address and transferring control to the next owner. The block ownership is determined statically by applying a hashing function on a block address. If higher-order bits are chosen for hashing, the block prediction accuracy goes down due to under-utilization of predictor

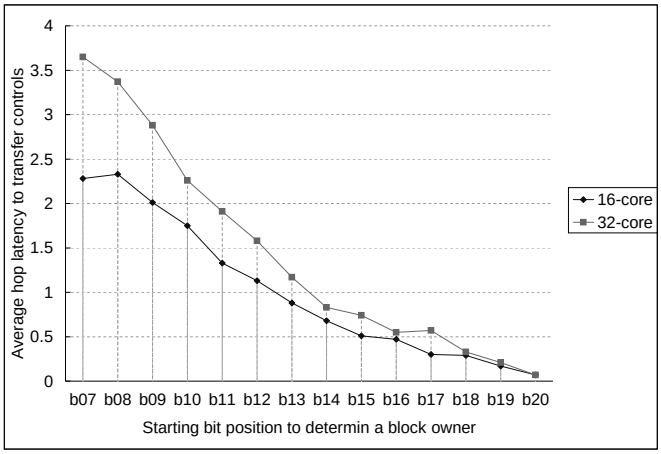


Figure 5.16: Average hop latency for control hand-off for 16-core and 32-core with various starting bit positions to determine a block owner

tables because too many blocks are mapped to few cores. On the other hand, using lower-order bits increases the control hand-off latency due to a frequent change of block owners. As shown in Figure 5.16 and Figure 5.15, using bits 12 through 15 achieves both a low misprediction rate and low handshaking overhead. The 32-core configuration shows similar trend but with greater variance in the hop latency. Because this selection of bits depends on the core configuration and potentially the program's characteristics, opportunities exist to adjust the hash function dynamically.

5.3.3 Operand Communication Overheads

Figures 5.17 and 5.18 show the two components that contribute to operand delivery latency: the network hop latency and the latency due to network contention. Both figures show the average delivery time for memory operands (*mem*) and the average delivery time for all operands (*all*). The memory operands include either operands transferred from a memory instruction to a destination cache bank or from a destination cache bank to a tar-

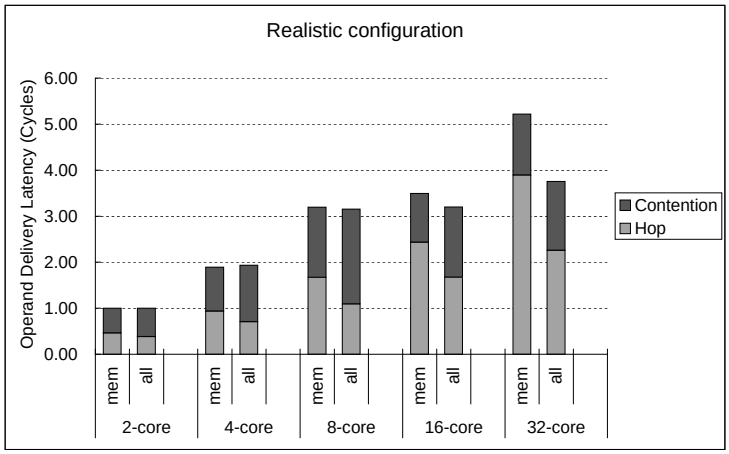


Figure 5.17: Average delivery times of memory operands and all operands : default

get instruction. The hop latency increases more sharply for memory operands than for *all*, which includes all operands. Because the compiler influences instruction placement, it can optimize for communication locality by placing dependent instructions on the same or nearby cores [21]. As shown in Figure 5.17, the compiler effectively reduces the operand delivery latency, producing an average hop latency ranging from 0.4 at two cores to 2.3 cycles at 32 cores. (As opposed to 3.9 cycles at 32 cores with no static instruction placement optimization)

Because memory addresses are not known until runtime, the compiler cannot optimize memory instruction placement. Thus the hop counts for memory operands depend on to which core's cache their addresses map. Figure 5.18 represents the average operand delivery time if memory scheduling could be made perfect, meaning that all memory operands are serviced at the local core's cache bank. Perfect memory scheduling reduces the operand delivery time between a memory instruction and a destination cache bank to zero, thereby

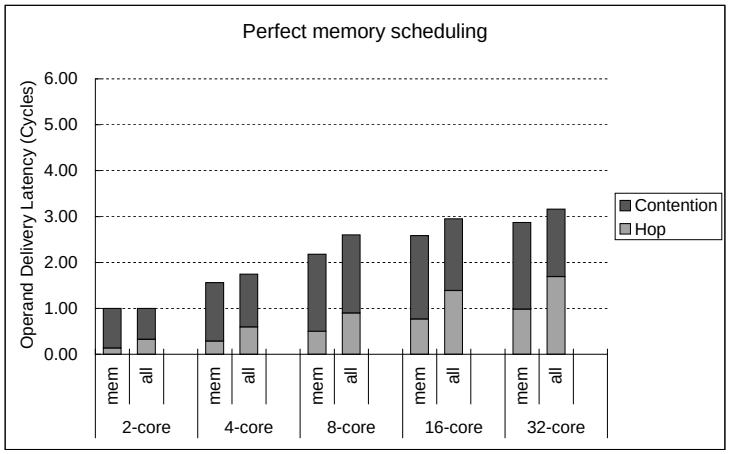


Figure 5.18: Average delivery times of memory operands and all operands : assuming ideal memory scheduling

reducing the overall hop latency to 1.7 cycles at 32 cores and produces a 7% performance improvement. This result demonstrates the potential benefits of compile-time memory disambiguation techniques which would allow better memory instruction alignment and scheduling.

High latency and/or low bandwidth channels between producer and consumer instructions spread across different cores can negatively impact performance on spatial architectures like TFlex, TRIPS or CMPs. To examine the criticality of operand delivery, we measure TFlex performance using different operand network configurations with 1-cycle and 2-cycle latency per hop across cores and with channels wide enough to communicate one (1x) and two data operands (2x) simultaneously. We also measure the performance with infinite bandwidth, which eliminates all network contention.

Figures 5.19 and 5.20 shows that the speedup for each of the above configurations over 1-core for varying number of cores and the best performing number of cores for both

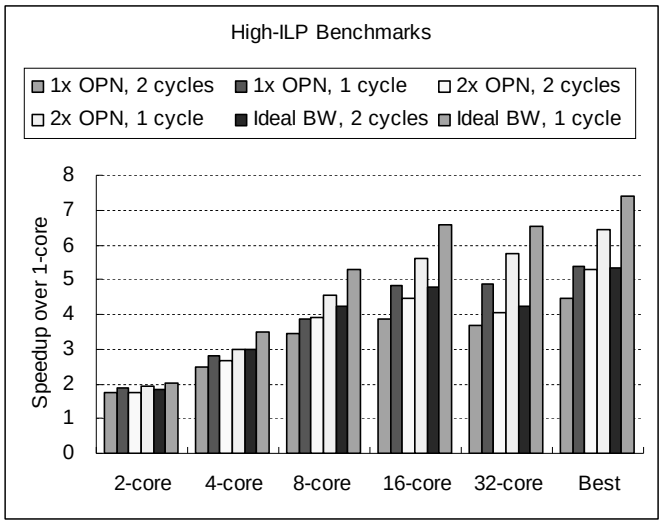


Figure 5.19: Operand network sensitivity analysis: High-ILP Benchmarks

high-and low-ILP benchmarks. The bar group *Best* represents the performance when each application is run with its own best-performing number of cores.

We observe that in the low-ILP benchmarks, the low operand delivery latency is crucial and the OPN with 2x bandwidth and 2-cycle latency per hop performs 18% worse than the OPN with 1x bandwidth and 1-cycle hop latency. For the high-ILP benchmarks, however, bandwidth is also crucial and the OPN with 2x bandwidth and 2-cycle latency per hop performs equivalent to the OPN with 1x bandwidth and 1-cycle hop latency. Intuitively, this is behavior is to be expected as benchmarks with more parallelism are likely to have more operands in-flight and benchmarks with low parallelism are likely to have dependence chains on the critical path.

When configured to the best-performing number of cores per application, doubling bandwidth for high-ILP benchmarks provides 17% performance improvement, while doubling bandwidth for low-ILP benchmarks shows only 4% improvement. On the other hand,

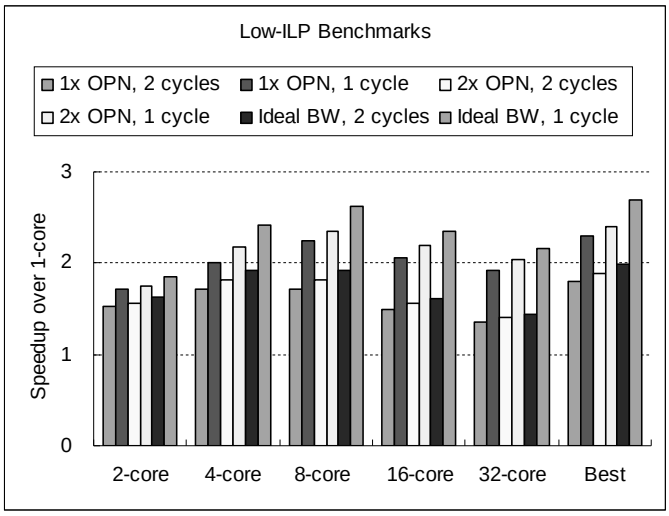


Figure 5.20: Operand network sensitivity analysis: Low-ILP Benchmarks

reducing the end-to-end operand hop latency is equally important both for high-ILP benchmarks and for low-ILP benchmarks, resulting in 22% difference for high-ILP benchmarks and 27% difference for low-ILP benchmarks between 1-cycle hop latency and 2-cycle hop latency.

Unsurprisingly, our sensitivity analysis illustrates the interesting interplay between available parallelism, latency and bandwidth. TFlex can adjust the number of cores to meet the concurrency needs of the application, even if it means using fewer processor to limit operand communication overheads.

5.3.4 Distributed Memory Disambiguation Overheads

As described in Section 5.2.5, the TFlex microarchitecture combines flush and NACK/retry features of the operand network to reduce the overhead of LSQ overflow. We evaluate a range of policies to determine when to re-issue a NACKed memory instruction in the

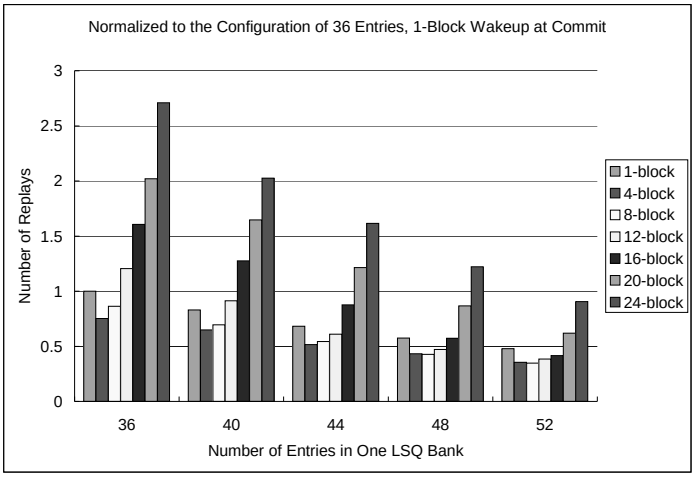


Figure 5.21: Number of LSQ replays normalized to the configuration of 36-entry, one-block wakeup at commit

issue window. Re-issuing instructions too soon (i.e. immediately upon NACK) can degrade performance by clogging the network, possibly re-generating multiple NACKs for the same instruction. Instead, our policy triggers re-issue when a non-speculative block commits, which is likely the right time since the overflowed LSQ bank may obtain available slots after block commit.

However, waking up all NACKed instructions simultaneously upon block commit can still bring the same negative effect as re-issuing instructions too soon. Conversely, constraining the timing for re-issue too much can limit the amount of instruction-level parallelism. To find the optimal policy, we vary the number of speculative blocks that contain NACKed instructions to wake up when a block commit signal is broadcasted.

For six of a total 26 benchmarks, the memory accesses are unevenly distributed and cause significant LSQ overflows on a 32-core TFlex architecture. Figure 5.21 shows the

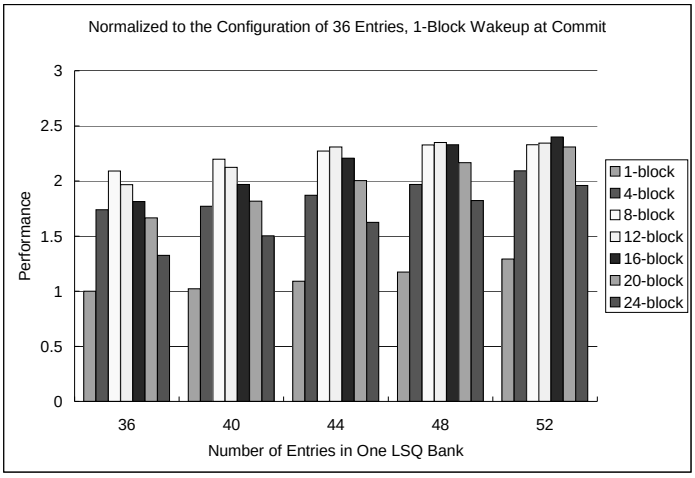


Figure 5.22: Performance normalized to the configuration of 36-entry, one-block wakeup at commit

number of re-issues in those six benchmarks normalized to the 36-entry LSQ bank with the policy of waking up NACKed instructions from one speculative block ahead.

The number of re-issues due to LSQ overflow can affect the power consumed by operand network and execution units and also affect the performance by causing network congestion (as shown in Figure 5.21). In general, as more NACKed instructions are woken up upon block commit, the possibility of re-generating overflow increases, resulting in more re-issues. Interestingly, we notice that the number of re-issues decreases when waking up instructions from one speculative block to four speculative blocks. This is because waking up more instructions can hasten the rate of committing blocks, which deallocates the occupied LSQ entries faster.

Figure 5.22 shows that as we wake up NACKed instructions from beyond a certain number of speculative blocks (i.e., 12 blocks at 44-entry LSQ bank), increasing the number

of re-issued instructions does not contribute to speeding up the rate of committing blocks any more and only congests the network, decreasing performance and increasing power consumption.

5.3.5 Level-2 Cache Organizations for TFlex

To understand how Level-2 cache designs affect performance, we compare the decoupled L2 and integrated L2 organization. As shown in Figure 5.10, a building block in the integrated L2 organization consists of two TFlex cores and a 256KB L2 cache. For comparison, we add the 4x OPN configuration in the integrated design since the total number of operand network routers is the half of those in the decoupled design. We use a two-cycle latency between hops in the integrated design due to the increased dimension of a building block. In Figure 5.23, we split the benchmarks into two categories along the amount of ILP, and normalize the performance of two integrated L2 configurations to the performance of the decoupled L2 design at various numbers of cores.

We first observe that the integrated L2 design outperforms the decoupled design at smaller processor configurations by offering the high bandwidth and low latency access to L2 caches. The smaller processor configurations are also more sensitive to L2 access latency due to the small issue window size. However, as more cores are aggregated, cores that belong to different building blocks are spaced farther apart, which affects the performance negatively. At both 16-core and 32-core configurations, the low-ILP benchmarks with the integrated L2 show 14% less performance than the decoupled design while the high-ILP benchmarks show almost no difference. However, TFlex can be configured with the best-performing number of cores per application, and the integrated L2 design can minimize the impact of increased hop latency by choosing a smaller processor configurations

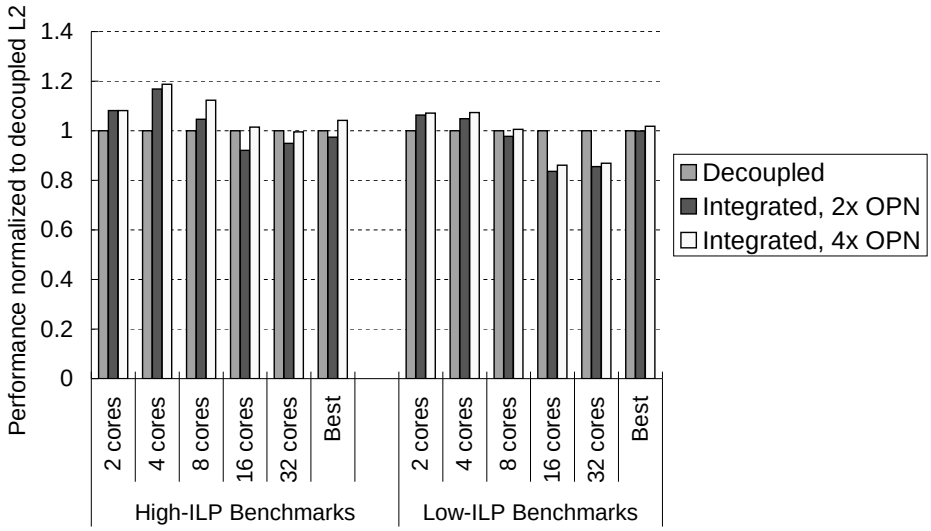


Figure 5.23: Performance comparison between the decoupled L2 design and the integrated L2 design

for low-ILP benchmarks, thus showing 5% improvement for the high-ILP benchmarks and 1% improvement for low-ILP benchmarks.

5.4 Comparison Across Configurations

In this section, we evaluate the TFlex processor with various number of cores and compare against a fixed-granularity TRIPS processor in terms of three different operating targets: performance, area efficiency, and energy efficiency.

5.4.1 Baseline

We choose the TRIPS processor as the baseline to compare against TFlex for three reasons. First, because TRIPS and TFlex share the same ISA and software infrastructure we can compare their microarchitectures without needing to compensate for ISA and system level artifacts. Second, TRIPS is a natural baseline because, unlike TFlex, TRIPS has limited options for supporting different processing granularities. For instance, TRIPS can only be configured either as an ILP engine supporting 1K in-flight instructions, or in an SMT mode with four threads each with maximum of 256 instructions per thread. We only compare against the single-threaded mode of TRIPS. TFlex can instead be configured for a range of granularities to adapt to various operating targets when the need arises. Finally, having access to the TRIPS hardware design and implementation provides a solid methodology for modeling TFlex, giving us higher confidence in the performance, area, and power estimates.

Baseline Validation: For TRIPS to be a satisfactory baseline, it must achieve at least a reasonable level of performance. To establish this baseline, we compare the performance of the TRIPS hardware to that of an Intel Core2 Duo system on the suite of EEMBC, SPEC, and hand-optimized benchmarks shown in Table 5.2. The Intel Core2 Duo measurements were taken on a Dell E520 system that has a 2.1GHz Intel Core2 Duo processor with 2GB 533 MHz DDR2 SDRAM memory. The TRIPS system has two TRIPS processors running at 366Mhz and 2GB DDR1 SDRAM memory running at 200 MHz. The C and Fortran codes for Intel Core2 Duo were compiled using gcc 4.1.2 -O3, and the PAPI 3.5.0 library was used to collect performance counter results [14]. For the TRIPS system also performance counters are used to read cycle counts. All experiments use only a single core in the Core2 and TRIPS systems, and we use cycle count only as the metric for comparing per-

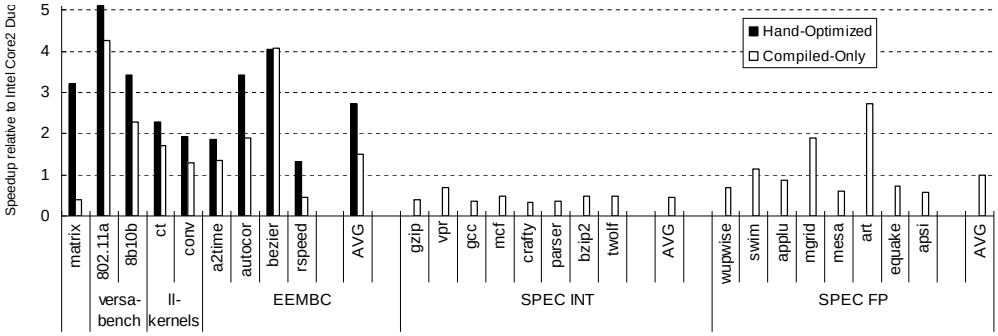


Figure 5.24: Relative performance (1/cycle count) for TRIPS normalized to Intel Core2 Duo.

formance to account for differences in process technology, design methodology, and size of the design team.

Figure 5.24 shows that on the hand-optimized benchmarks TRIPS uniformly outperforms the Core2 and achieves an average 2.7x speedup [1]. For the compiled benchmarks, TRIPS is approximately 50% faster on average than the Core2 on versabench, ll-kernels and EEMBC benchmarks, 3% worse on SPEC FP and 57% worse on SPEC INT. Ongoing work on the TRIPS compiler promises to close the gap from hand-optimized code as the hand-optimizations were performed with compiler automation in mind and include tuning loop unroll counts and eliminating false load/store dependence with enhanced register allocation.

Simulator Validation: The simulator used in this study can model both the TRIPS hardware prototype and the TFlex microarchitecture since they both use the same ISA, and also have similar functional components such as on-chip interconnection networks, caches, ex-

[1] For matrix multiplication, we use the optimized binary from GotoBLAS [38] for Intel Core2 Duo and we compare the FPC (FLOPS/cycle) instead of cycle counts.

ecution units, and register files. The simulator was validated by simulating a configuration similar to the TRIPS prototype hardware and comparing the cycle counts against the actual hardware on a set of EEMBC benchmarks and microbenchmarks extracted from the SPEC 2000 suite. We observe that the cycle count estimates from the simulator are within 7% of the hardware cycle counts. The results, presented in the subsequent sections, indicate that TFlex performance improvements are much greater than the modeling error.

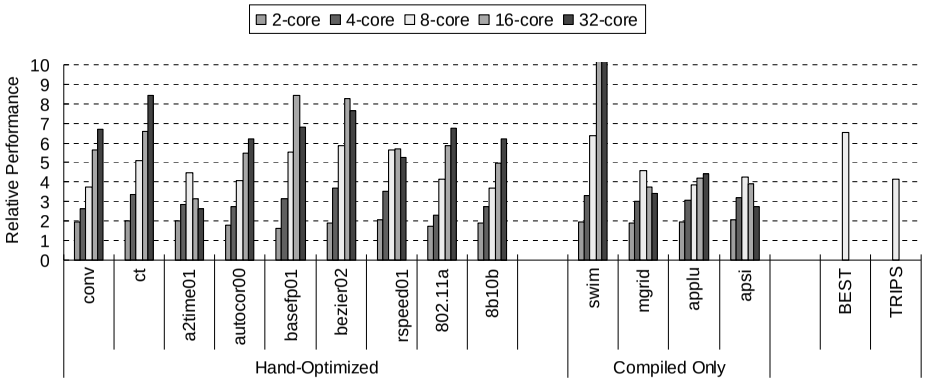
The simulated baseline TRIPS microarchitecture matches that described by Sankaralingam et al. [99] with the exception that the L2 capacity of the simulated TRIPS processor is 4MB to enable a fair comparison with TFlex.

The TFlex architecture also includes two microarchitecture optimizations that could be applied to improve the baseline performance and area efficiency of the TRIPS processor. First, the bandwidth of the operand network is doubled to reduce contention and improve performance. Second, TFlex cores are dual-issue, as opposed to the single-issue execution tiles in TRIPS. For a comparable issue width processor, dual issue improves area efficiency by reducing the number of FPUs without changing the peak integer issue bandwidth.

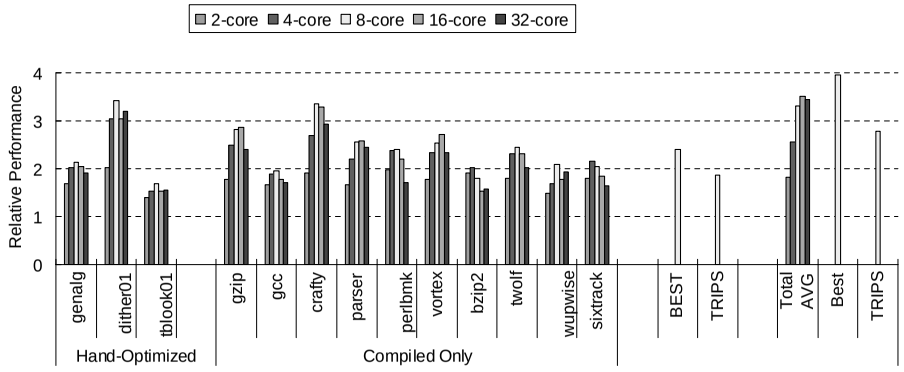
Our results show that TFlex can be configured to have optimal processor ganularities depending on different application characteristics and operating targets.

5.4.2 Performance Comparison

Figure 5.25a and b show the performance of the TRIPS prototype architecture and that of TFlex configurations ranging from 2 to 32 cores, normalized to the performance of a single TFlex core. The 26 benchmarks on the x-axis are arranged into categories of low and high IPC. On average, the 16-core TFlex configuration performs best and shows 3.5x speedup over a single TFlex core. When the processor is configured to the best performing number



(a) High-ILP Benchmarks



(b) Low-ILP Benchmarks

Figure 5.25: Performance of different applications running on 2 to 32 cores on a CLP normalized to a single TFlex core

Structures		Single TFlex core		8 TFlex cores		Single TRIPS core	
	Subcomponent	Size	Area	Size	Area	Size	Area
Fetch.	Block Predictor	8Kbit		64Kbit		64Kbit	
	I-Cache	8KB	1.36	64KB	10.88	80KB	7.66
Register Files		128 entries	0.81	1K entries	6.47	512 entries	3.04
Exec.	Issue Window	128 entries		1K entries		1K entries	
	ALU	INT(2) FP(1)	2.95	INT(16) FP(8)	23.6	INT(16) FP(16)	39.36
Primary D-Cache	D-Cache	8KB	3.48	64KB		32KB	
Subsystem	LSQ	44 entries		352 entries	27.84	1K entries	33.44
Routers			0.88		7.04		11
Sum			9.48		75.83		94.5

Table 5.3: Microarchitecture parameters and area estimates (mm^2)

of cores for each application (represented by the bar "BEST"), the performance of TFlex increases an additional 13% and the overall speedup over a single TFlex core reaches 4x. These results indicate that, using the proposed execution model, sequential applications can be effectively run across multiple cores to achieve substantial speedups.

On average, an 8-core TFlex, which has the same area and issue width as the TRIPS processor, outperforms TRIPS by 17%, reflecting the benefits of additional operand network bandwidth as well as twice the L1 cache bandwidth stemming from fine-grained distribution of the cache. The best TFlex configuration outperforms TRIPS by 42% demonstrating that adapting the processor granularity to the application granularity provides significant improvements in performance.

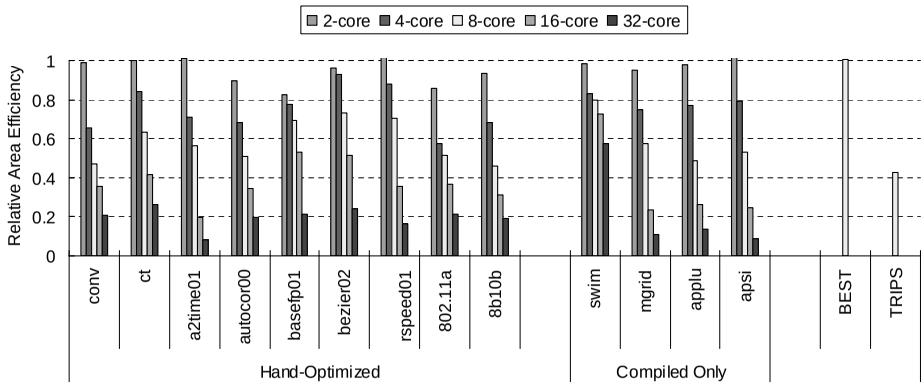
Considering the increased resources within each core, we also ran simulations with two cycles per hop in the operand network. When configured to the best performing number of cores per application, TFlex with the two-cycle hop latency lost performance by 22%, reducing the speedup over TRIPS to 19%.

5.4.3 Area Efficiency Comparison

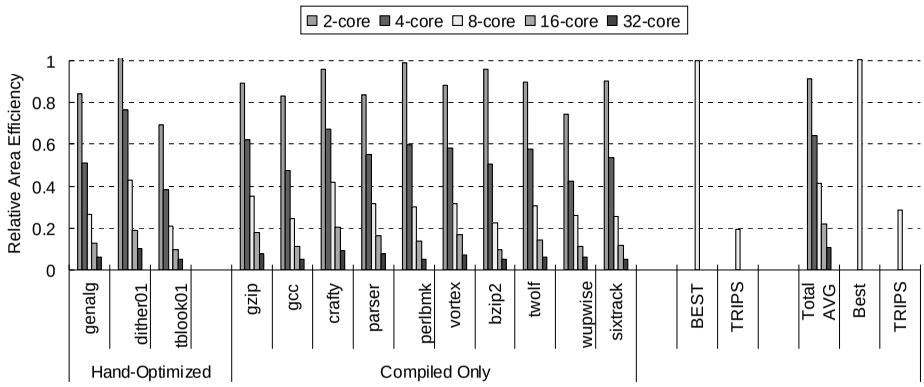
We examine the area required for a TFlex processor and the performance/area as a function of the number of cores. The area of each microarchitectural component in a single TFlex core was estimated from the post-synthesis netlist (before final place-and-route) of the 130nm ASIC implementation of the TRIPS prototype. Table 5.3 presents the area of different microarchitectural components in a single TFlex core and a single TRIPS core. In the table, we also present the 8-core TFlex configuration that is the largest configuration that can fit in a single TRIPS core area. We excluded the L2 cache, memory controller, and peripheral devices to compare the area of the two processors. Table 5.3 shows that a single TFlex core is approximately ten times smaller than a single TRIPS core. Since the TRIPS chip contains dual TRIPS cores in the 130nm ASIC technology, the same die area has enough room to allow 16 TFlex cores. A 130nm, 18mm x 18mm die can accommodate 8 TFlex cores with 1.5MB of L2 cache. Assuming linear scaling, a 32-core TFlex array with 4MB of L2 cache would fit comfortably on a 12mm x 12mm die at 45nm.

Figure 5.26a and b plot the performance per area ($1/(\text{cycles} \times \text{mm}^2)$) for the TRIPS processor and various TFlex configurations, all normalized to a single TFlex core. If ample threads are available, the performance/area metric is equivalent to a throughput metric. For most benchmarks, area efficiency peaks either at one or two cores; beyond two cores (four-wide issue), performance improvements scale at a slower rate than area growth. On average, TFlex can produce up to 3.4 times better performance/area than TRIPS. The fixed, aggressive processor configuration in TRIPS (16-wide issue, 1K issue window) pays a higher penalty in terms of performance/area, especially for low-ILP benchmarks and results in around 5x degradation in performance/area.

Unlike a fixed-granularity architecture, a composable architecture can balance area



(a) High-ILP Benchmarks



(b) Low-ILP Benchmarks

Figure 5.26: Performance per unit area for different applications running on 2 to 32 cores on TFlex CLP normalized to single-core TFlex

efficiency versus peak performance demand depending on various runtime factors including the number of active threads.

5.4.4 Power Efficiency Comparison

In this section, we compare the power dissipation of the baseline TRIPS processor with that of various TFlex configurations. First, we present the power modeling methodology and next, we present the results.

Power Modeling Methodology

I collaborated with Madhu Saravana Sibi Govindan to derive a power model for both the TRIPS and the TFlex processors. To estimate the power consumed by the TFlex processor, we estimated the power of three constituent components: the clock power, the DIMM power, and the core power which includes everything on chip excluding clock and the DIMM power. We estimated the core power and the clock power from the TRIPS hardware netlist. Since TFlex and TRIPS use similar microarchitectural building blocks, we were able to obtain capacitance estimates for scaled microarchitectural structures from TRIPS. The activity factors for each of these structures were obtained from the cycle-accurate simulator. The clock tree power for TFlex was estimated by measuring the clock power for the TRIPS implementation and scaling it down to one TFlex core. The clock power was then scaled linearly according to number of TFlex cores. The DIMM power was estimated using analytical power models from Micron [56] and by counting the number of off-chip accesses in the simulator. Since the baseline TRIPS processor does not have clock-gating, our TFlex power models do not support clock-gating to enable a fairer comparison. However, we did implement a simple clock-gating power model to examine how the optimal number of

Structures	8 TFlex cores		Single TRIPS core	
	High-ILP	Low-ILP	High-ILP	Low-ILP
Fetch. (Block Predictor, I-cache)	1.07 (4.2%)	1.24 (5.3%)	0.91 (3.1%)	1.06 (3.9%)
Exec. (Reg, issue window, ALUs)	3.04 (12.0%)	1.44 (6.1%)	2.94 (10.0%)	1.25 (4.5%)
L1 D-cache subsystem (D-cache, LSQ, MSHR)	0.57 (2.2%)	0.33 (1.4%)	0.59 (2.0%)	0.29 (1.0%)
Operand Network Routers	0.22 (0.9%)	0.36 (1.2%)	0.30 (1.1%)	
L2, DIMM, I/O	3.33 (24.2%)	3.34 (14.2%)	3.46 (11.7%)	3.33 (12.2%)
Clock tree	14.89 (58.5%)	14.89 (63.2%)	18.39 (62.4%)	18.39 (67.1%)
Leakage	2.06 (8.1%)	2.06 (8.7%)	2.81 (9.5%)	2.81 (10.2%)
Sum	25.45 (100%)	23.54 (100%)	29.45 (100%)	27.42 (100%)

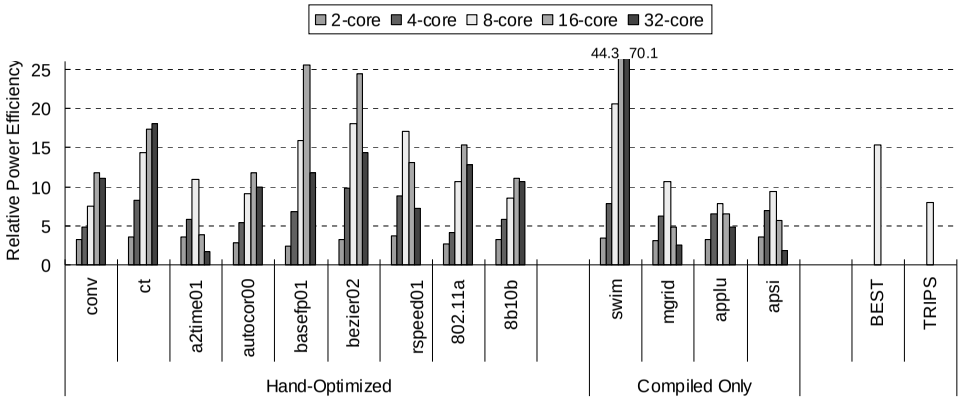
Table 5.4: Sample Power Breakdown (Watt) for High-ILP and Low-ILP Benchmarks

TFlex cores for a given benchmark changes. We validated the power measurements by configuring the TFlex simulator in the TRIPS mode and comparing the reported power against the power measured from the hardware system. The power difference between two models for the benchmark suite was less than 10%.

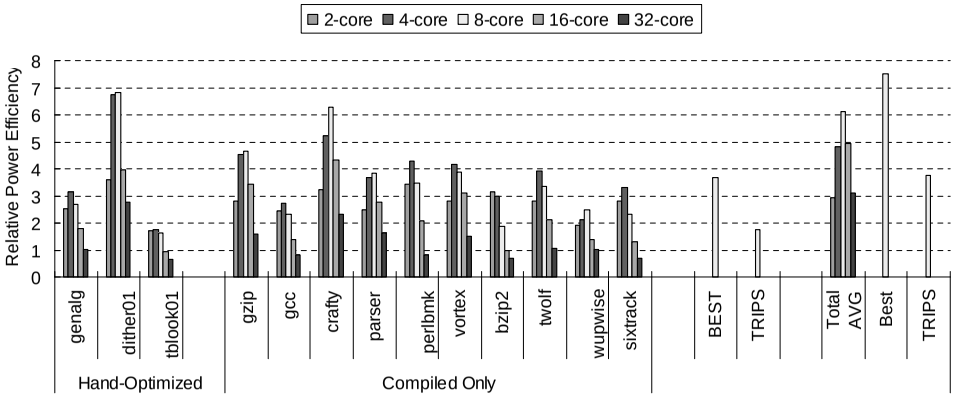
Power Estimation Results

Table 5.4 breaks down the average of total power dissipation of all benchmarks into various categories like Fetch, Execution, L1 D-Cache, Routers, L2 cache/DIMM/IO, the clock tree and leakage power. The benchmarks are categorized into two groups; high-ILP benchmarks and low-ILP benchmarks. The power dissipated in the individual categories are relatively small because the clock tree power in all these categories has been reported as a separate category in the table. We show this breakdown for 8 TFlex cores and a single TRIPS core.

We use the performance^2/Watt metric to assess the overall energy efficiency - this metric accounts for the power efficiency of the architecture scaled by the time taken to execute the benchmark [36]. Figure 5.27a and b show performance^2/Watt metric for the various TFlex configurations and the TRIPS configuration over all the benchmarks. On average,



(a) High-ILP Benchmarks



(b) Low-ILP Benchmarks

Figure 5.27: Performance^2/Watt for different applications running on 2 to 32 cores on TFlex CLP normalized to single-core TFlex - without clock gating

the results show that TFlex with 8 cores performs the best in terms of performance^2/Watt metric. The most power-efficient TFlex configuration ranges from 4 to 32, with 8 cores being the best overall fixed configuration. The flexibility to choose the best one on a per-application basis produces an overall average improvement of about 22% over any fixed TFlex system. The power efficiency of a fixed 8-core TFlex system is about 64% better than a fixed TRIPS system. Although both have the same execution bandwidth, TRIPS has twice the power-hungry floating-point units, but which are not used every cycle. Fine-grained clock gating of these FPUs could improve the relative power efficiency of TRIPS.

Clock-Gating Results

As previously mentioned, the above results assume that both TFlex and TRIPS do not support clock-gating - all latches are switching every cycle - to make a fair comparison with TRIPS. We also did an experiment with a simple clock-gating model for the TFlex cores - the model assumes that when all the units of a TFlex core are idle, all of them are clock-gated except the operand network routers (to enable routing the operands to other TFlex cores), the L2 subsystem and the on-chip network routers. As shown in Figure 5.28, the clock-gating results indicate that the overall power efficiency increases with clock gating, but the optimal number of cores did not change from the experiments without clock-gating - TFlex with 8 cores were the optimal configurations in terms of performance^2/Watt both with and without clock-gating.

5.4.5 Ideal Operating Points

The most important capability of the CLP approach is not the absolute benefit over the alternatives at any operating targets, but the ability to shift to different operating points

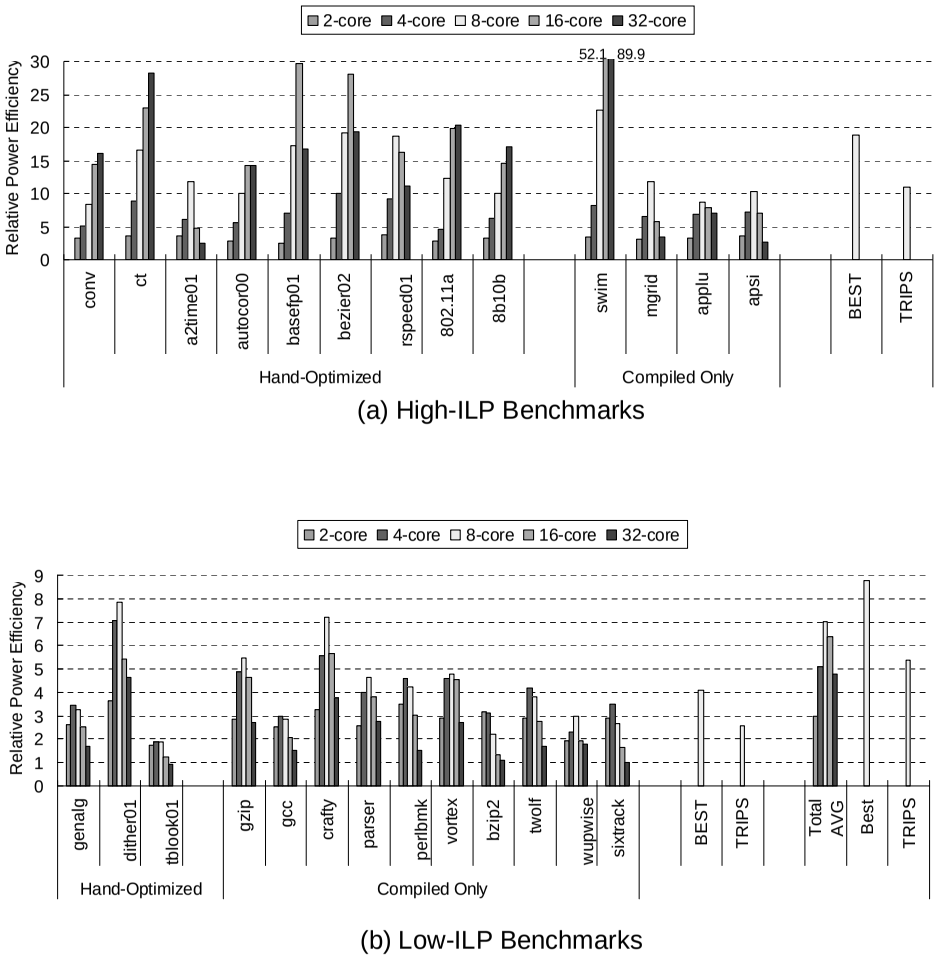


Figure 5.28: Performance^2/Watt for different applications running on 2 to 32 cores on TFlex CLP normalized to single-core TFlex - with clock gating

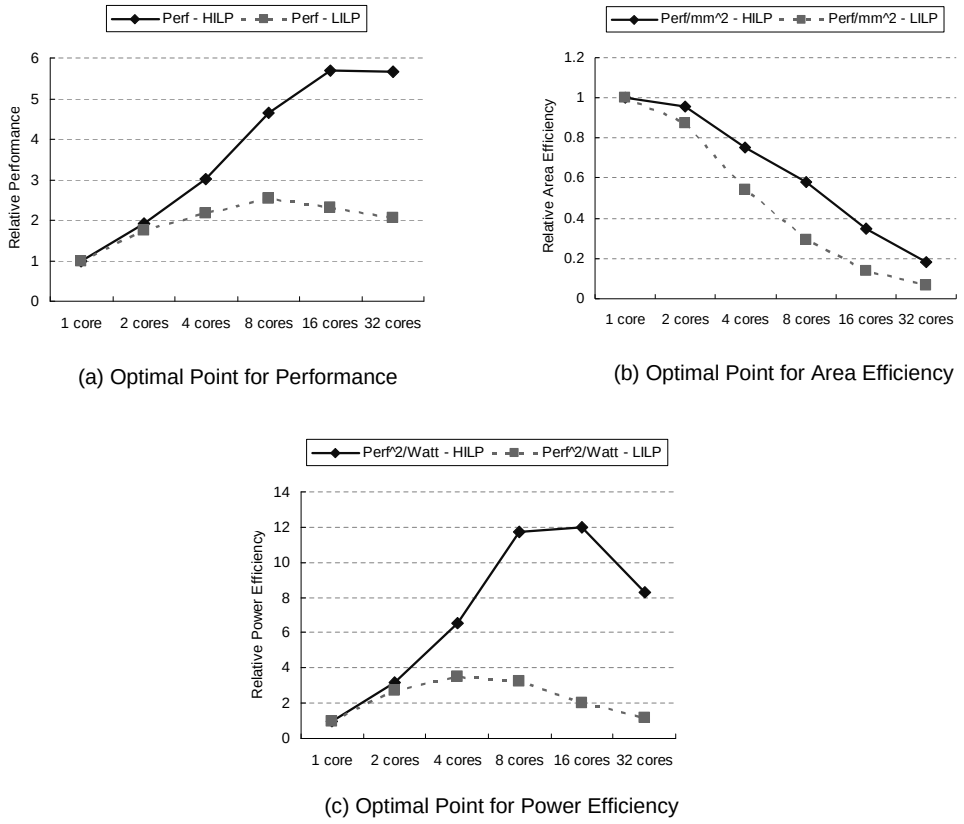


Figure 5.29: Optimal point at different operating targets

when the need arises. Figure 5.29 and Table 5.5 shows the optimal points depending on application characteristics and operating targets. We plot the performance, performance per area, and the performance^{2}per watt, categorized by the high- and low-ILP benchmark groups. The graph shows how each metric varies across different composed processor sizes, while the table shows the number of cores at which each metric is maximal.

Depending on which metric is most important—raw performance, area efficiency, or energy efficiency—the best configuration is quite different, ranging from one core per thread for maximum area efficiency, eight cores per thread for maximum energy efficiency,

Metrics	High-ILP # of cores	Low-ILP # of cores	All apps. # of cores	Fixed Ratio to the TRIPS	Best Ratio to the TRIPS
Perf	16	8	16	1.2	1.4
Perf/mm^2	1	1	1	3.4	3.4
Perf^2/Watt	8	8	8	1.6	2.0
Perf^3/Watt/mm^2	8	4	4	2.8	4.1

Table 5.5: Optimal point at different operating targets

to 16 for maximum performance. Moreover, if the system can identify the best operating point at an application-specific granularity, additional improvements are possible. An interesting open question is whether further improvements can be obtained by exploiting coarse-grain program phases using dynamic reconfiguration while the applications are running.

5.5 Summary

In this chapter we have described a CLP (Composable Lightweight Processor) that provides microarchitectural support for run-time configuration of fine-grained CMP processors, allowing flexibility in aggregating cores together to form larger logical processors. A disadvantage of this approach is that it relies on non-traditional ISA support, using EDGE architectures rather than RISC or CISC. An advantage is that unlike prior work, the larger logical processor groups together distributed resources to form unified logical resources, including instruction sequencing, memory disambiguation, data caches, instruction caches, register files, and branch predictors. That grouping permits higher performance than previous distributed approaches (such as thread-level speculation) as well as a finer degree of configurability.

Since most future performance gains will come from concurrency, future systems will need to mine concurrency from all levels. Depending on the workload mix and number of available threads, the right place to find the concurrency will likely change frequently for general-purpose systems, rendering the design-time freezing of processor granularity in traditional CMPs a highly undesirable option. A CLP permits the run-time system to make informed decisions about how to go about exploiting concurrency, whether it be from a single thread running on many distributed cores, or many threads running on partitioned resources. Other factors that may affect the resource configuration include power/performance trade-offs and the amount of concurrency within each thread.

Chapter 6

Conclusions

Clock rate scaling can no longer sustain computer system scaling due to power and thermal constraints, diminishing performance returns of pipeline scaling [45, 51], and process variation [13]. Future performance improvement must therefore come primarily from mining concurrency from applications. Unfortunately, conventional approaches will be problematic, as increasing global on-chip wire delays will limit amount of state available in a single cycle, thereby hampering the ability to mine concurrency.

To address these technology challenges, industry has migrated to chip multiprocessors in the hope that software threads will provide the concurrency needed for future performance gains. However, relying on compilers or programmers to parallelize applications has had only limited success over the past years and may result in disrupting software development productivity in the future. Moreover, Amdahl's law dictates that the sequential portions of execution will eventually hamper the overall performance growth.

Another disadvantage of conventional CMP architectures is their relative inflexibility, making any fixed CMP designs ill-suited to meet various application demands and

operating targets. Application domains have become increasingly diverse, now spanning desktop, network, server, scientific, graphics, and digital signal processing. In each domain, applications have different granularity of concurrency and memory requirements. Even within the same application, the amount of computational requirement and the size of working set differ across various execution phases.

In current designs, the granularity (e.g., issue width), the number of processors on each chip, and the memory hierarchies (e.g., cache capacity of each level) are fixed at design time based on the target workload mix. Once deployed, however, the ideal balance between the granularity, the number of cores per chip, and capacity of each level may change as the workload mix changes. While parallel (TLP-centric) workloads favor a processor design with many small cores, the "inherently sequential" (ILP-centric) workloads take good advantage of a few, but large, aggressive cores. There is also trade-off between larger, slower caches for applications with large working sets and smaller, faster caches for applications that are less memory intensive. These diverse characteristics of workloads render the design-time freezing of granularity in traditional processor and cache architectures an undesirable option.

In this dissertation, we explored the concept of the *composability*, for both processors and on-chip memories, to address both the increasing wire delay problem and the inflexibility of conventional CMP architectures for meeting various application demands. The basic concept of composability is the ability to adapt to diverse applications and operating targets by aggregating fine-grained processing units or memory units.

6.1 Summary

This dissertation identifies four main principles for a composable architecture.

- Composable architectures employ a distributed substrate consisting of multiple fine-grained processing and memory units. The fine-grained units are inherently more power-efficient and achieve technology scalability with respect to future global wire delay increases.

- Composable architectures provide the ability (1) to aggregate fine-grained units to compose into a larger logical unit and (2) to match each application to the composed logical unit best suited to meet its performance, power, and throughput demands.

- The number of fine-grained units combined to execute each application can be dynamically changed transparently to the running application.

- Composable architectures provide an ISA and microarchitectural support to combine distributed fine-grained units in a power- and area- efficient manner. The overheads to support composability in a distributed substrate should be minimized.

This dissertation evaluates composable architectures that have two main components: (1) NUCA (Non-Uniform Access Cache Architectures) and (2) CLP (Composable Lightweight Processors)

6.1.1 NUCA (Non-Uniform Access Cache Architecture)

The current designs of large level-2 caches will not work effectively in future wire-delay dominated technologies. This dissertation describes a fundamentally new class of cache design, called Non-Uniform Access Level-2 Cache Architecture (NUCA). NUCA caches break large caches into many banks that are independently accessible with a switched network embedded in the cache. Lines can be mapped into this array of memory banks with

fixed mappings, as in the static NUCA organization (S-NUCA), or dynamic mappings (D-NUCA), where cache lines can move around within the cache.

Adaptivity for various working set sizes: This dissertation shows that by gradually migrating cache lines within the cache nearer to the processor as they are used, the bulk of accesses go to banks close to the processor. The working set thus clusters in the banks closest to the processor, so long as the working set is smaller than the cache, resulting in hit latencies considerably lower than the average access latency to a bank. Because of its adaptability, the D-NUCA eliminates the trade-off between larger, slower caches for applications with large working sets and smaller, faster caches for applications that are less memory intensive.

Composability for various memory organizations: Applications from different domains have different memory access patterns. While applications that have irregular access patterns will favor the cache design, streaming applications from scientific and graphics domain will take good advantage of a scrachpad memory. As a proof of concept, we built a composable secondary memory system in the TRIPS prototype. The TRIPS secondary memory system is composable, as it consists of multiple, aggregable memory banks, which can be configured differently. The possible memory organizations include a 1MB L2 cache or a 1MB scratchpad memory or any mix between them totalling 1MB.

Extension of NUCA to CMP Level-2 Caches: In this dissertation, we extended the concept of NUCA to CMP Level-2 caches and explored the well-known design trade-offs between a private L2 design with lower hit latency and a shared L2 design with larger effective cache capacity. The proposed L2 cache substrate can support a flexible sharing

degree from low-latency, private logical caches, to highly shared caches, or any intermediate design point between the two. We show that the L2 hit latency more than doubles for a fully shared cache compared to private caches and makes a larger sharing degree less effective; despite the benefits of eliminating many of the off-chip misses. Then, we explored a dynamic mapping policy to address slow access time in a highly shared cache. On the 16-processor CMP design that we evaluated, we observed only modest performance gains over the S-NUCA design with the best sharing degree. The overhead of searching data in the D-NUCA design degrades performance significantly. Therefore, we conclude that the performance gains of the D-NUCA design are unlikely to justify the added design complexity. However, for a subset of applications we observed that the dynamic data migration capabilities of D-NUCA can reduce the average hit latency, driving the ideal sharing degree to higher sharing degrees. In addition, D-NUCA showed the potential benefit of reducing energy consumption by decreasing the on-chip network traffic in higher sharing degrees. Based on our observation of when dynamic mapping works, dynamic mapping could be a more attractive alternative to static mapping as the number of processor cores and L2 cache capacities increase. However, inventing a less complex search mechanism in D-NUCA will be the key enabler for adopting D-NUCA designs in future CMP caches.

6.1.2 CLP (Composable Lightweight Processor)

A CLP consists of a large number of low-power, lightweight processor cores that can be aggregated dynamically to form more powerful logical single-threaded processors. Compared to conventional CMP architectures that have a “rigid” granularity, CLPs provide flexibility to dynamically allocate resources to different types of concurrency, ranging from running a single thread on a logical processor composed of many distributed cores, to running many

threads on separate physical cores. The system can also use energy and/or area efficiency as metrics to choose the configurations best suited for any given point.

ISA support for Composability: While composability can also be provided using traditional ISAs [54], we examined CLPs in the context of a block-based Explicit Data Graph Execution (EDGE) architectures [15], that provide many benefits over traditional ISAs.

The EDGE ISA has the following two key features: (1) explicit specification of producer-consumer relationship between dependent instructions (2) block-atomic execution of hyperblocks. These two features alleviate the need for power hungry hardware structures like associative register renaming and issue windows.

This dissertation shows that the above mentioned features of the EDGE ISA make it attractive for providing efficient composability as well. Since the dataflow graph is statically and explicitly encoded in the instruction stream, it is simple to shrink or expand the graph on fewer or larger number of execution resources as desired with virtually no additional hardware. Further, the coordination overheads required to run a single thread application on multiple cores can be significantly reduced if the unit of coordination is a block of instructions rather than individual instructions. Using the EDGE ISA, we developed an implementation of CLP, named "TFlex".

Microarchitecture for Composability: The microarchitectural structures in a composable processor must allow their capacity to be incrementally increased or decreased as the number of participating cores increase or decrease.

To provide this capability we identify and repeatedly apply two principles. First, the hardware resources should not be oversized or undersized to suit either a large processor configuration or a small configuration. Second, we avoid physically centralized microar-

chitectural structures completely. Decentralization allows the size of structures to be grown without the undue complexity traditionally associated with large centralized structures. We evaluate the overheads to support composibility in a distributed substrate and show that the TFlex microarchitecture keeps these overheads sufficiently low.

Configuration for an Ideal Operating Point: This dissertation demonstrates that the best processor configuration is quite different depending on application characteristics and operating targets (metric) — raw performance, area efficiency, or power efficiency. The TFlex microarchitecture provides the ability to shift to different processor configurations when the need arises.

Scaling Degree of Composition: In this dissertation we explored a range of compositions (i.e. degree of composition) — from two to 32 cores — to synthesize a logical processor to run a single-threaded application. We found that aggregating cores beyond 16 does not yield enough benefits to justify the additional resources. Even in terms of raw performance, we observed that only a few benchmarks (from the high-ILP group) showed reasonable benefits beyond 16 cores. Moreover, the ideal configurations for both area- and power-efficiency were achieved at a much smaller number of cores. The main reason is that the performance penalty due to the increased number of hops outweighs the benefits from exploiting higher concurrency. Prior research has also shown that operand communication latency is the primary bottleneck for scaling single-threaded performance in a distributed architecture [83]. While we present a composable architecture for efficiently mining parallelism from a contiguous program region, we believe that solutions composing greater than 16 cores for achieving even higher performance from single-threaded application must extract concurrency from non-contiguous regions of the program.

6.2 Final Thoughts

A composable architecture aggregates fine-grained processing units or memory units into larger logical units and provides the ability to adapt to different application demands and various operating targets. This dissertation opens up two broad challenges for future work.

Finding an optimal point: While this dissertation presents composable architectures that can adapt to different application demands and various operating targets, the detailed mechanisms on how to find an optimal point are not explored. We envision multiple methods of controlling the allocation of cores to threads. Compilers can provide hints on the amount of ILP by analyzing the whole program statically and performing off-line profiling. Depending on the granularity of configuration and the number of threads involved, the following two approaches can be considered: hardware-based decisions and software-based decisions. Hardware-based decisions can respond more quickly to catastrophic thermal events or adapt to fine-grained intra-thread phase diversity. On the other hand, the software-based approach (possibly by the operating system) can involve multiple threads and introduce more complex scheduling algorithms considering time, space, job priority, and other operating targets (e.g., energy).

There is much related work on optimal job scheduling. Several papers [31, 110] focused on SMT/CMP job scheduling that aims at attaining optimal throughput. Recent scheduling work on heterogeneous CMP demonstrates the benefits of mapping each job to the core that most closely matches the resource demands of the application [9, 68]. Besides pure performance, energy-aware job scheduling [25] takes power or energy into consideration to make scheduling decisions. Techniques such as DVFS or thread migration can be used to enable energy-aware scheduling [35, 80, 122].

The configurability of a composable architecture offers another degree of freedom when balancing power and performance. However, too much freedom does not necessarily produce the best policy because of possible state-space explosion. The right schedule should be both workload- and operating target-dependent and be able to dynamically adjust itself to discover the optimal point.

Finding the best balance between ILP and TLP: There are many advantages to building a future CMP out of fine-grained small processor cores. First, small processor cores are inherently more power efficient because of lower capacitance in the active state associated with physically small layout and wires. The finer-granularity control by DVFS provides further opportunities to optimize power consumption. Second, small cores produce higher performance per unit area for parallel software. Finally, the low design complexity compared to designing a large core is a significant advantage. However, the criticality of single-thread application performance and Amdahl's law will hamper adoption of smaller processor cores in current CMP architectures. We envision that composability proposed in this dissertation could open the door to adopting smaller processor cores in future CMPs.

The best way to exploit many small cores is to extract thread-level parallelism (TLP) from applications. Generally, exploiting TLP (throughput) is a more power-efficient way of obtaining performance than exploiting ILP (scalar) [41]. More applications in the future therefore are anticipated to be written in multi-threaded fashion [116], with support from programming languages [18] or hardware/software mechanisms to ease concurrent programming [47]. However, balancing between ILP and TLP in multi-threaded applications will pose a great challenge due to the following reasons. First, applications have different characteristics in terms of granularity of parallelism: Some applications from scientific, media, and server domain are more amenable to extracting TLP while other applications

are inherently sequential. Second, many parallel applications are incrementally parallelized to amortize the programming effort over time, and thus present different amounts of TLP depending on stages of development [54]. Finally, the optimized parallel software for one CPU generation may not produce the optimized performance for each successive generation of CPUs as the number of integrated cores in a chip is expected to keep increasing. We envision that future CMPs should be flexible and reconfigure themselves to perform best amidst various amount of ILP and TLP existing in applications by considering the amount of ILP per thread, thread synchronization overheads, and reconfiguration cost. Looking forward, a composable architecture will further blur the distinction between conventional uniprocessors and multiprocessors, which we view as a promising direction.

Appendices

Appendix A

Comparison Between Hand-Optimized And Compiled Code

In Section 5.4.2, we reported the performance of TFlex with the two different benchmark suites: a hand-optimized suite and a compiler-generated suite.

Figure A.1 shows the performance of kernel benchmarks (seven EEMBC benchmarks, three LL kernels, two Versa benchmarks) before and after hand optimizations. For reference, the figure also shows the performance with the perfect configuration in which performance is only constrained by issue width, using perfect block prediction, perfect memory disambiguation and zero-cycle operand delivery with unlimited bandwidth.

Unsurprisingly, the difference between the real and the perfect configuration is less for the hand-optimized benchmarks (compared to the compiler-optimized benchmarks) and for the low-ILP benchmarks (compared to the high-ILP benchmarks). The right side of Fig-

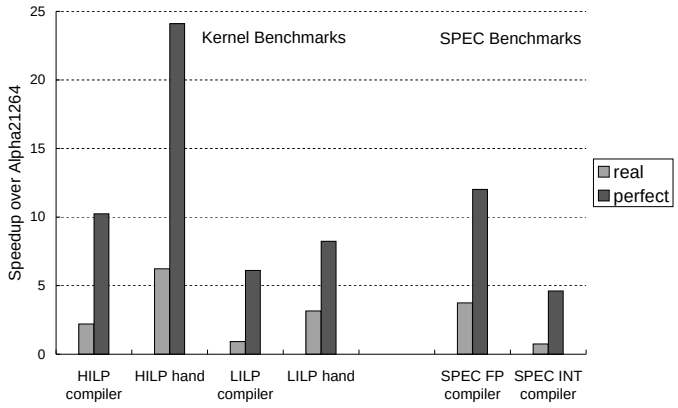


Figure A.1: Performance comparison between compiler-optimized and hand-optimized applications under the baseline configuration and the perfect configuration

ure A.1 shows the performance of SPEC benchmarks with only compiler optimizations (we did not hand-optimize the SPEC benchmarks). We found that the SPEC floating-point and integer benchmarks follow trends similar to the high-ILP and low-ILP kernel benchmarks, respectively.

Our hand-optimizations for the benchmarks did not involve a complete rewrite, but instead focused on optimizations that we expect the compiler to perform well:

- Instruction merging when the same instructions are predicated on both true and false predicates.
- Tuning the loop unroll counts.
- False load/store dependence elimination with enhanced register allocation.

The performance gap between the compiled code and the hand-optimized code is about 3x in both the high- and the low- ILP benchmarks. This gap indicates the potential optimization opportunities for the compiler.

Appendix B

Area Comparison with the Alpha 21264

In Section 5.4.3, we evaluated the area efficiency with various number of cores in TFlex and compared against the TRIPS processor. We reported that at 130nm, a 18mm x 18mm die can integrate 8-TFlex cores with 1.5MB of L2 cache. Assuming linear scaling, at 45nm on a 12mm x 12mm die a 32-core TFlex with 4MB cache seems feasible. Compared to conventional out-of-order issue CMP architectures, TFlex can integrate more cores in a chip.

To analyze the area benefits against a conventional out-of-order superscalar processor core, we estimate the area of each microarchitectural component in the Alpha 21264 processor core by using the published die sizes and the die photograph [61]. The die size of the Alpha 21264 is reported to be 310mm^2 at 350nm. To compare the Alpha 21264 core and the single TFlex core at the same technology, we scale the area of each component to a 65nm technology. In the Alpha, we apply a 10% reduction to account for die photo

Structures	Alpha scaled (mm^2 at 65nm)	TFlex scaled (mm^2 at 65nm)	Alpha uarch	TFlex uarch
Fetch (I-cache + ITLB + BP)	2.65 (29%)	0.35 (15%)	64KB I-cache	8KB I-cache
Register File	1.02 (11%)	0.17 (7%)	10-port 232 entries	2-port 128 entries
Renaming and Issue Window	1.30 (14%)	0.21 (9%)	35-entry CAM	128-entry RAM (no renaming)
Functional Units	1.28 (14%)	0.60 (26%)	4-INT ALU, 2 FP	2-INT ALU, 1FP
D-cache	1.94 (21%)	0.45 (19%)	64KB D-cache	8KB D-cache
LSQ + DTLB + Miss Handling	0.86 (10%)	0.36 (16%)	2 32-entry CAMs	1 40-entry CAM
Routers	N/A	0.19 (8%)		
Sum	9.04 (100%)	2.32 (100%)		

Table B.1: Area comparison between the Alpha 21264 and a single TFlex core

measurement errors. Since a custom implementation of TRIPS would be smaller than an ASIC implementation, we apply a 40% area reduction to random logic in TFlex and leave the SRAM/register arrays untouched. Finally, we add a 10% area increase to both the Alpha and TFlex to reflect estimation errors in our linear process technology scaling model.

Table B.1 shows that a single TFlex core is approximately four times smaller than the Alpha 21264 in 65nm. To first order, this ratio is reasonable, since a TFlex core has 1/8th the instruction and data cache capacity and half the number of ALUs. The major area advantages in TFlex (aside from smaller caches) come from eliminating complex out-of-order structures such as a per-instruction register renamer, an associative issue window, and multi-ported register files, each of which is obviated by the TRIPS ISA and execution model. For example, the 10-ported register files in the Alpha are about six times larger than the dual-ported TFlex register file, even though the total number of entries is only twice that of TFlex. In addition, the RAM-structured issue window in TFlex is six times smaller than the CAM-based window in the Alpha, even with four times the number of

issue window entries. Based on the results in Table B.1, in a 65nm process, a 32-core TFlex microarchitecture with 4MB L2 cache could be implemented in only 144mm^2.

Bibliography

[1] Kartik K. Agaram, Stephen W. Keckler, Calvin Lin, and Kathryn S. McKinley. Decomposing memory performance: data structures and phases. In *Proceedings of the 5th International Symposium on Memory Management*, pages 95–103, 2006.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] David H. Albonesi, Rajeev Balasubramonian, Steve Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(12):49–58, 2003.

[4] D.H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 248–259, December 1999.

[5] Murali Annavaram, Ed Grochowski, and John Paul Shen. Mitigating Amdahl's law

through EPI throttling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 298–309, 2005.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[7] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 218–229, 2001.

[8] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002 Revision 2, NASA Ames Research Laboratory, Mountain View, CA, August 1991.

[9] Saisanthosh Balakrishnan, Ravi Rajwar, Michael Upton, and Konrad K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 506–517, 2005.

[10] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, 2002.

[11] Bradford M. Beckmann and David A. Wood. TLC: Transmission line caches. In *Pro-*

ceedings of the 36th Annual International Symposium on Microarchitecture, pages 43–54, 2003.

[12] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 319–330, 2004.

[13] Shekhar Y. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[14] Shirley Browne, Jack Dongarra, N. Garner, Kevin S. London, and Philip Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 42, 2000.

[15] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, Calvin Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[16] Ramon Canal, Joan-Manuel Parcerisa, and Antonio González. A cost-effective clustered architecture. In *Proceedings of the 8th International Symposium on Parallel Architectures and Compilation Techniques*, pages 160–168, 1999.

[17] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 264–276, June 2006.

[18] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-

oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[19] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 55–66, 2003.

[20] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 357–368, 2005.

[21] Katherine E. Coons, Xia Chen, Doug Burger, Kathryn S. McKinley, and Sundeep K. Kushwaha. A spatial path scheduling algorithm for edge architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[22] Willian James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, Inc, 2004.

[23] James C. Dehnert, Brian Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing - Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st Annual International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[24] Rajagopalan Desikan, Doug Burger, Stephen W. Keckler, and Todd M. Austin. Sim-

alpha: A validated execution-driven Alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, University of Texas at Austin, 2001.

[25] Matt Devuyst, Rakesh Kumar, and Dean Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *International Parallel and Distributed Processing Symposium*, 2006.

[26] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 233–, 2002.

[27] Pradeep Dubey. A platform 2015 workload model: Recognition, mining and synthesis moves computers to the era of tera. Technical report, Intel, February 2005.

[28] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1), 1995.

[29] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isacc Hernandez, Toni Juan, Geoff Lowney, Mathew Mattina, and Andre Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of The 29th International Symposium on Computer Architecture*, pages 281–292, May 2002.

[30] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multi-

cluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, December 1997.

[31] Alexandra Fedorova, Margo I. Seltzer, Christopher Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Annual Technical Conference, General Track*, pages 395–398, 2005.

[32] Joseph A. Fisher, Paolo Faraboschi, and Giuseppe Desoli. Custom-fit processors: Letting applications define architectures. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 324–335, December 1996.

[33] David Flynn. AMBA: Enabling Reusable On-Chip Designs. *IEEE Micro*, 17(4):20–27, 1997.

[34] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 230–239, 2001.

[35] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 260–270, 2004.

[36] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. IEEE Journal of Solid-State Circuits, 31(9):1277–1284, September 1996.

[37] Ricardo E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, /2000.

[38] Kazushige Goto and Robert van de Geign. On reducing tlb misses in matrix multiplication. Technical Report TR-02-55, Department of Computer Sciences, The University of Texas at Austin, 2002.

[39] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W. Keckler, and Doug Burger. Implementation and Evaluation of On-Chip Network Architectures. In *IEEE International Conference on Computer Design*, 2006.

[40] Paul Gratz, Karthikeyan Sankaralingam, Heather Hanson, Premkishore Shivakumar, Robert McDonald, Stephen W. Keckler, and Doug Burger. Implementation and evaluation of a dynamically routed processor operand network. In *Proceedings of the 1st International Symposium on Networks-on-Chip*, pages 7–17, 2007.

[41] Ed Grochowski, Ronny Ronen, John Paul Shen, and Hong Wang. Best of both latency and throughput. In *IEEE International Conference on Computer Design*, pages 236–243, 2004.

[42] E.G. Hallnor and S.K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, June 2000.

[43] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael K. Chen, and Kunle Olukotun. The stanford hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[44] Reiner W. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE*, pages 642–649, 2001.

[45] A. Hartstein and Thomas R. Puzak. The optimium pipeline depth for a microprocessor. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 7–13, May 2002.

[46] Allan Hartstein and Thomas R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 117–128, 2003.

[47] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[48] J.M. Hill and J. Lachman. A 900MHz 2.25 MB cache with on-chip CPU now in Cu SOI. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 171–177, February 2001.

[49] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[50] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[51] M.S. Hrishikesh, Keith Farkas, Norman P. Jouppi, Doug Burger, Stephen W. Keckler, and Premkishore Sivakumar. The optimal logic depth per pipeline stage is 6 to 8 fo4

inverter delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 14–24, May 2002.

[52] Jaehyuk Huh. *Hardware Techniques to Reduce Communication Costs in Multiprocessors*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, May 2006.

[53] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future CMPs. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, September 2001.

[54] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martnez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[55] J. Rubinstein, P. Penfield, and M.A. Horowitz. Signal delay in RC tree networks. *IEEE Transactions on Computer-Aided Design*, CAD-2(3):202–211, 1983.

[56] J. W. Janzen. DDR SDRAM Power Calculation Sheet. Micron, 2001.

[57] Roy M. Jenevein and James C. Browne. A control processor for a reconfigurable array computer. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 81–89, 1982.

[58] Norman P. Jouppi and Steven J. E. Wilton. An enhanced access and cycle time model for on-chip caches. Technical Report TR-93-5, Compaq WRL, July 1994.

[59] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), September 2005.

[60] Richard E. Kessler. *Analysis of Multi-Megabyte Secondary CPU Cache Memories*. PhD thesis, University of Wisconsin-Madison, December 1989.

[61] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[62] Richard E. Kessler, Mark D. Hill, and David A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 43(6):664–675, June 1994.

[63] Richard E. Kessler, Richard Jooss, Alvin R. Lebeck, and Mark D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 131–139, May 1989.

[64] Chankyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[65] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Computers*, 48(9):866–880, 1999.

[66] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, 2003.

[67] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip mul-

tiprocessing. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 195–206, 2004.

[68] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 64–75, 2004.

[69] Fernando Latorre, José González, and Antonio González. Back-end assignment schemes for clustered multithreaded processors. In *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 316–325, 2004.

[70] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35–44, 1990.

[71] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, New York, NY, USA, 1998. ACM Press.

[72] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *Proceedings of the 34th Annual International symposium on Computer Architecture*, June 2007.

[73] Jian Li and José F. Martínez. Power-performance considerations of parallel comput-

ing on chip multiprocessors. *ACM Transactions on Architecture and Code Optimization*, 2(4):397–422, 2005.

[74] Haiming Liu. Hardware techniques to improve cache efficiency, Ph.D proposal, April 2007.

[75] Bertrand A. Maher, Aaron Smith, Doug Burger, and Kathryn S. McKinley. Merging Head and Tail Duplication for Convergent Hyperblock Formation. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, December 2006.

[76] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[77] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[78] Robert McDonald, Doug Burger, Stephen W. Keckler, Karthikeyan Sankaralingam, and Ramadass Nagarajan. TRIPS processor reference manual. Technical Report TR-05-19, Department of Computer Sciences, The University of Texas at Austin, March 2005.

[79] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[80] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *SIGOPS Oper. Syst. Rev.*, 40(4):403–414, 2006.

[81] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: a power/performance/thermal view. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 177–186, 2006.

[82] Trevor N. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, 2001.

[83] Ramadass Nagarajan. *Design and Evaluation of a Technology-Scalable Architecture for Instruction-Level Parallelism*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, May 2007.

[84] Ramadass Nagarajan, Karthikeyan Sankaralingam, Stephen W. Keckler, and Doug Burger. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 40–51, December 2001.

[85] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[86] Basem A. Nayfeh, Kunle Olukotun, and Jaswinder Pal Singh. The impact of shared-cache clustering in small-scale shared-memory multiprocessors. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 74–84, 1996.

[87] A. Nicolau and J. Fisher. Measuring the parallelism available for very long word architectures. *IEEE Transactions on Computers*, 33(11):968–974, November 1984.

[88] H. Pilo, A. Allen, J. Covino, P. Hansen, S. Lamphier, C. Murphy, T. Traver, and

P. Yee. An 833MHz 1.5w 18Mb CMOS SRAM with 1.67Gb/s/pin. In *Proceedings of the 2000 IEEE International Solid-State Circuits Conference*, pages 266–267, February 2000.

[89] Timothy Mark Pinkston and Jeonghee Shin. Trends toward on-chip networked microsystems. *International Journal of High Performance Computing and Networking*, 3(1):3–18, 2005.

[90] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 90–101, 2001.

[91] M.D. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 54–65, December 2001.

[92] Steven A. Przybylski. *Performance-Directed Memory Hierarchy Design*. PhD thesis, Stanford University, September 1988. Technical report CSL-TR-88-366.

[93] R. M. Rabbah, I. Bratt, K. Asanovic, and A.Agarwal. Versatility and versabench: A new metric and a benchmark suite for flexible architectures. Massachusetts Institute of Technology Technical Report MIT-LCS-TM-646, June 2004.

[94] Paul Racunas and Yale N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 22–31, 2003.

[95] Nitya Ranganathan. Control flow speculation for distributed architectures, Ph.D proposal, April 2007.

[96] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 214–224, 2000.

[97] Karthikeyan Sankaralingam. *Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, October 2006.

[98] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 422–433, May 2003.

[99] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, Madhu Saravana Sibi Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethmadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 480–491, December 2006.

[100] Matthew C. Sejnowski, Edwin T. Upchurch, Rajan N. Kapur, Daniel P. S. Charlu, and G. Jack Lipovski. An overview of the Texas reconfigurable array computer. In *AFIPS Conference Proceedings*, pages 631–642, 1980.

[101] The national technology roadmap for semiconductors. Semiconductor Industry Association, 2001.

[102] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable memory disambiguation for high ilp processors. In *36th International Symposium on Microarchitecture*, pages 399–410, December 2003.

[103] Simha Sethumadhavan, Franziska Roesner, Joel S Emer, Doug Burger, and Stephen W. Keckler. Late-Binding: Enabling unordered load-store queues. In *Proceedings of the 34th Annual International symposium on Computer Architecture*, June 2007.

[104] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[105] Premkishore Shivakumar and Norman P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation, August 2001.

[106] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 2–13, 2003.

[107] Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn S. McKinley. Compiling for EDGE architectures. In

Fourth International ACM/IEEE Symposium on Code Generation and Optimization (CGO), March 2006.

[108] Aaron Smith, Ramadass Nagarajan, Karthikeyan Sankaralingam, Robert McDonald, Doug Burger, Stephen W. Keckler, and Kathryn S. McKinley. Dataflow Predication. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*, December 2006.

[109] James E. Smith and Gurindar S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[110] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[111] Kimming So and Rudolph N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–109, July 1988.

[112] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[113] Gurindar S. Sohi and Manoj Franklin. High-performance data memory systems for superscalar processors. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[114] Evan Speight, Hazim Shafi, Lixin Zhang, and Ram Rajamony. Adaptive mechanisms

and policies for managing cache hierarchies in chip multiprocessors. In *Proceedings of the 32nd Annual International symposium on Computer Architecture*, June 2005.

[115] Standard Performance Evaluation Corporation. *SPEC Newsletter*, Fairfax, VA, September 2000.

[116] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[117] David Tarjan, Shyamkumar Thoziyoor, and Norman Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Labs, 2006.

[118] Michael Bedford Taylor, Walter Lee, Saman P. Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 341–353, February 2003.

[119] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman P. Amarasinghe, and Anant Agarwal. Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 2–13, 2004.

[120] Joel M. Tendler, J. Steve Dodson, J. S. Fields Jr., Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[121] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading:

Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[122] Vibhore Vardhan, Daniel Grobe Sachs, Wanghong Yuan, Albert F. Harris, Sarita V. Adve, Douglas L. Jones, Robin H. Kravets, and Klara Nahrstedt. Integrating fine-grain application adaptation with global adaption for saving energy. In *Proceedings of the 2nd International Workshop on Powe-Aware Real-Time Computing (PARC)*, 2005.

[123] Kenneth M. Wilson and Kunle Olukotun. Designing high bandwidth on-chip caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 121–132, June 1997.

[124] Steven J. E. Wilton and Norman P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[125] Alexander Wolfe. "Intel Clears Up Post-Tejas Confusion", May 2004. http://www.crn.com/it-channel/18842588.

[126] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[127] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 336–345, June 2005.

[128] Hongtao Zhong, Steven Lieberman, and Scott Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proc. 2007 International Symposium on High Performance Computer Architecture*, February 2007.

Vita

Changkyu Kim was born in Seoul, Korea on August 26th 1973, the son of Taewon Kim and Inja Yum. After graduating from Seoul Science High School, he entered Seoul National University in 1993. He received a Bachelor of Science degree in Computer Engineering from Seoul National University in February 1997, followed by a Master of Science degree in Computer Engineering in 1999. In the fall of 2000, he joined the doctoral program at the Department of Computer Sciences at the University of Texas at Austin.

Permanent Address: Seocho Gu, Banpo Dong
MIDO APT 303-1211,
Seoul, Korea, 137-044

This dissertation was typeset with LaTeX 2_ε by the author.