Copyright
by
Behnam Robatmili
2011

The Dissertation Committee for Behnam Robatmili
certifies that this is the approved version of the following dissertation:

Efficient Execution of Sequential Applications on Multicore Systems

Committee:

Doug C. Burger, Supervisor

Kathryn S. McKinley, Supervisor

Stephen W. Keckler

Calvin Lin

Steve Reinhardt

Efficient Execution of Sequential Applications on Multicore Systems

by

Behnam Robatmili, 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 2011

Dedicated to my wife Sahar.

Acknowledgments

I wish to thank the multitudes of people who helped me along the way to completing this dissertation. I thank my committee, Doug Burger , Kathryn McKinley, Steve Keckler, Calvin Lin, and Steve Reinhardt, for their valuable feedback on my research. I am particularly indebted to my advisor, Doug Burger for his supervision of this work. I also thank several colleagues and friends who helped me along the way to completing this work including Katie Coons, Bert Maher, Dong Li, Hadi Esmaeilzadeh, Sibi Govindan, Jeff Diamond, Boris Grot, and Mark Gebhart.

Efficient Execution of Sequential Applications on Multicore Systems

Publication No. ____________

Behnam Robatmili, Ph.D.
The University of Texas at Austin, 2011

Supervisors: Doug C. Burger
Kathryn S. McKinley

Conventional CMOS scaling has been the engine of the technology revolution in most application domains. This trend has changed as in each technology generation, transistor densities continue to increase while due to the limits on threshold voltage scaling, per-transistor energy consumption decreases much more slowly than in the past. The power scaling issues will restrict the adaptability of designs to operate in different power and performance regimes. Consequently, future systems must employ more efficient architectures for optimizing every thread in the program across different power and performance regimes, rather than architectures that utilize more transistors. One solution is composable or dynamic multicore architectures that can span a wide range of energy/performance operating points by enabling multiple simple cores to compose to form a larger and more powerful core.

Explicit Data Graph Execution (EDGE) architectures represent a highly scalable class of composable processors that exploit predicated dataflow block execution and distributed microarchitectures. However, prior EDGE architectures suffer from several energy and performance bottlenecks including expensive intra-block operand communication due to fine-grain instruction distribution among cores, the compiler-generated fanout trees built for high-fanout operand delivery, poor next-block prediction accuracy, and low speculation rates due to predicates and expensive refills after pipeline flushes. To design an energy-efficient and flexible dynamic multicore, this dissertation employs a systematic methodology that detects inefficiencies and then designs and evaluates solutions that maximize power and performance efficiency across different power and performance regimes. Some innovations and optimization techniques include: **(a) Deep Block Mapping** extracts more coarse-grained parallelism and reduces cross-core operand network traffic by mapping each block of instructions into the instruction queue of one core instead of distributing blocks across all composed cores as done in previous EDGE designs, **(b) Iterative Path Predictor (IPP)** reduces branch and predication overheads by unifying multi-exit block target prediction and predicate path prediction while providing improved accuracy for each, **(c) Register Bypassing** reduces cross-core register communication delays by bypassing register values predicted to be critical directly from producing to consuming cores, **(d) Block Reissue** reduces pipeline flush penalties by reissuing instructions in previously executed instances of blocks while they are still in the instruction queue, and **(e) Exposed Operand Broadcasts (EOBs)** reduce wide-fanout instruction overheads by extending the ISA to employ architecturally exposed low-overhead broadcasts combined with dataflow for efficient operand delivery for both high- and low-fanout instructions.

These components form the basis for a third-generation EDGE microarchitecture called T3. T3 improves energy efficiency by about 2× and performance by 47% compared to previous EDGE architectures. T3 also performs in a highly power efficient manner across a wide spectrum of energy and performance operating points (low-power to high-performance), extending the domain of power/performance trade-offs beyond what dynamic voltage and frequency scaling offers on state-of-the-art conventional processors. This high level of flexibility and power efficiency makes T3 an attractive candidate for future systems which need to operate on a wide range of workloads under varying power and performance constraints.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Future systems will support different types of workloads in a power efficient manner by employing heterogeneous processors on the same chip. For example, throughput applications such as graphics run on highly optimized throughput processors such as GPUs [3, 4]. Control- and memory-intensive threads, however, run on CPUs. CPUs traditionally employ power scaling methods to achieve power efficiency. However, the traditional power scaling methods such as dynamic voltage and frequency scaling (DVFS) are becoming less effective given the current trends of transistor scaling [36, 93]. The reason is partially related to the fact that as maximum supply voltage has declined over the several years, minimum supply voltage has almost remained constant [93]. This shrinking operating voltage ranges highly reduces achievable DVFS ranges. There have been several proposals in recent years to achieving energy-proportional computing for control-intensive and single-thread codes. An *asymmetric multicore* (A-CMP) [47] has a collection of cores of different execution granularities. For example, an A-CMP can have a few high-performance cores for running single-threaded code and several light-weight cores for running parallel code. A-CMPs can work efficiently for some types of workloads but they are not flexible enough to adapt to a wide range of workload characteristics. This lack of flexibility is due to the fixed issue width and execution bandwidth of the large cores allocated to sequential codes. One other alternative is to use architectural innovations to distribute execution of each thread across variable number of processing cores in a flexible manner [33, 36, 39, 41, 93]. Such dynamic distributed microarchitectures, which are called composable or dynamic multicores, can operate at different energy and performance operating points without relying on traditional DVFS methods. Additionally, to meet power constraints, such systems have to rely on microarchitecture or ISA features to achieve high energy and performance efficiency at each unique power/performance operating point.

Among those dynamic architectures, Explicit Data Graph Execution (EDGE) [75] architectures were conceived with the goal of enabling energy-efficient high performance, by distributing computation across simple tiles. By raising the level of control abstraction to an atomic predicated multi-exit block of instructions, in which branches are converted to predicates, control overheads such as branch prediction and commit can be amortized. By incorporating dataflow semantics into the ISA, aggressive out-of-order execution is possible while using less energy than RISC or CISC designs. The intra-block data-flow encodings push much of the run-time dependence graph construction to the compiler, reducing the energy required to support out-of-order execution through construction and traversal of those graphs. To date, EDGE architectures have not yet demonstrated these potential advantages [32].

1.1 Dissertation Contributions

This dissertation examines inefficiencies in early EDGE microarchitectures such as TRIPS and TFlex. We use a critical path systematic analysis for detecting inefficiencies and reducing these bottlenecks. Guided by this approach, this dissertation proposes the T3 EDGE microarchitecture [69] by re-inventing several important microarchitectural and ISA components in previous designs. This new design eliminates the issues associated with previous microarchitectures and maximizes power and performance efficiency for different power and performance regimes. Most of these innovations reduce critical delay and energy consumption of the system simultaneously, thus raising the energy efficiency of the entire system.

1.1.1 T3: An Energy-Efficient Dynamic Uniprocessor

This dissertation proposes few novel microarchitectural and ISA components for a third-generation EDGE microarchitecture called T3. These new features include:

- **Block mapping:** Intra-block dataflow communication across cores is a major bottleneck in early EDGE architectures. The *deep block mapping* mechanism [67] reduces operand network traffic and saves energy by mapping each block to the instruction queue of one core. Although this mechanism limits the intra-block parallelism to the issue width of each core, it improves inter-block parallelism by removing fine-grained network traffic. Deep mapping reduces the cross-core communication traffic by 50%, resulting in major delay and energy savings.

- **Cross-core register bypassing**: Another network bottleneck in early EDGE architectures is the inter-block cross-core register communication delay. The *Register bypassing* mechanism [68] reduces cross-core register communication delay by bypassing critical register values directly from producing to consuming cores. To predict critical cross-core communications, this mechanism employs a low-overhead distributed framework called *Distributed Block Criticality Analyzer* [68] (DBCA) that exploits different types of criticality information collected at block boundaries.

- **Pipeline flush handling**: Supporting large distributed instruction windows, the EDGE architectures suffer from pipeline flush delay and energy penalties when mispredictions occur. The *Block reissue* mechanism [68] reduces pipeline flush penalties and saves fetch and decode energy by allowing previously executed instances of a block to be reissued while they are still in the instruction queue. This mechanism halves the number of energy-hungry fetch and decode operations and also reduces the flush delay in the block pipeline.

- **Branch prediction**: The combination of speculative block-based execution and predication within blocks in EDGE architectures moves branch prediction off the critical path and alleviates the fetch bandwidth bottleneck. However, performing multi-exit next block prediction on each block results in loss of prediction accuracy as the global history of branches no longer includes those branches that have been converted into predicates. Additionally, the branches that are converted to predicates are evaluated at the execution stage rather than being predicted, thus manifesting themselves as execution bottlenecks. This dissertation proposes a mechanism called Iterative Path Prediction (IPP) [69] that quickly predicts an approximate multi-bit predicate path through an instruction block, appending that path to the global history to predict the next-block target address. The predicted path is then used to speculatively execute the predicates within the block, thus incorporating both predicate and branch target prediction in one microarchitectural component. By maximizing the speculation rate while increasing speculation accuracy at both block and instruction levels, this mechanism harvests 15% increase in performance and 5% core-wide energy savings when composing 16 cores to run each thread, as compared to TFlex [41].

- **Operand delivery**: The other problem with early EDGE designs such as TRIPS and TFlex is associated with operand delivery. The use of dataflow communication among instructions in each block eliminates the need for a broadcast bypass network, associative tag matching, and the register renaming logic found in conventional out-of-order processors. However, for high-fanout operands, the compiler must generate trees of move instructions to fanout values to destination instructions. These fanout instructions increase execution delay and consume additional energy. To address this issue, this dissertation employs a mechanism called Exposed Operand Broadcasts (EOBs) [69] proposed by Li et al. [48, 69] and explains how to integrate EOBs into T3. This mechanism exposes a small number of per-block broadcast identifiers to the compiler, which assigns them to the highest-fanout operands. For the operands using the architecturally visible broadcasts, a narrow distribution network conveys those operands to their consumers, eliminating most of the move instructions, and consuming little energy to distribute the high-fanout operands.

Exploiting these low-overhead features, the T3 microarchitecture maximizes energy and performance efficiency by saving execution latency and power consumption at the same time. We compare the performance and energy efficiency of T3 against previous EDGE architectures. On SPEC CINT2000, T3 increases average performance appreciably (over 47% with eight composed cores) while simultaneously reducing the energy consumed (27% with eight cores), which translates to about 2x improved energy delay product, as compared to TFlex.

We also examine the performance/power flexibility of T3 by comparing it to real conventional platforms by using both hardware measurements [25] and analytical power models [49]. For high-performance (10∼30 watts range) and low-energy references (1∼3 watts range), we use an Intel Core 2 and an Intel Atom processors, respectively. With low core counts (one or two), T3 consumes energy in the low-energy region while performing close to the high-performance region. When running with four or more composed cores per thread, T3 improves performance significantly while it consumes energy below the high-performance region. This degree of flexibility and energy efficiency allows T3 to explore power/performance trade-offs beyond those of conventional processors.

1.1.2 Systematic Bottleneck Detection and Reduction

Bottleneck analysis and removal is a challenging task for designing distributed systems due to their increased complexity. For example, distributed uniprocessors such as T3 try to merge multiple independent cores, transparent to software, to accelerate single-threaded workloads. This dissertation proposes a methodology that exploits critical path analysis for systematically analyzing and reducing the performance and scalability bottlenecks of such fully distributed processors [68]. In each optimization step, this method uses criticality information at two levels to focus optimization mechanisms efficiently,

A system-level breakdown of critical cycles reveals the contribution of each micro-architectural component. For each detected bottleneck, a fine-grained component-level breakdown indicates the scenarios under which the corresponding component turns into a bottleneck. This fine-grain information is then used to choose the right optimization method for the system and the process repeats. Figure 1.1 illustrates the iterative bottleneck analysis and

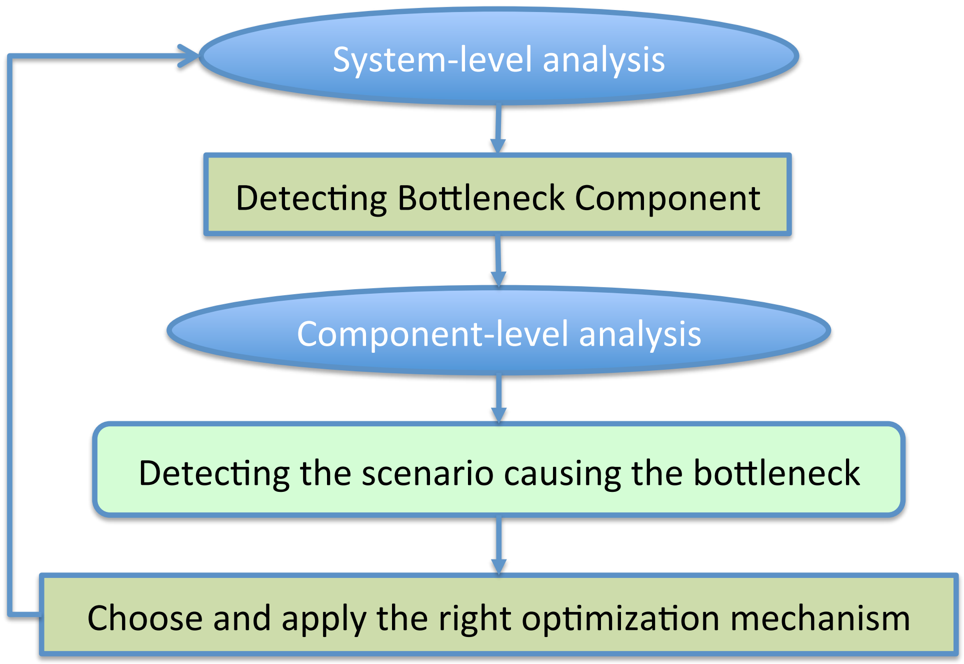

System-level analysis
Detecting Bottleneck Component
Component-level analysis
Detecting the scenario causing the bottleneck
Choose and apply the right optimization mechanism

Figure 1.1: Iterative bottleneck analysis and reduction methodology.

reduction proposed by this dissertation.

1.2 Dissertation Organization

This dissertation is organized as follows. Chapter 2 reviews related work. We focus on a few distinct areas. Distributed (composable) uniprocssors is the key area of our related work. We then focus on the related work associated with each individual mechanism used by the T3 microarchitecture including instruction mapping, instruction reuse, predicate prediction, hybrid operand delivery, and register bypassing between distributed cores. Finally, we discuss the prior work on critical path analysis.

Chapter 3 presents a background on EDGE and early EDGE architectures, TRIPS and TFlex, and their strengths and issues. The chapter then gives a short overview on the T3 microarchitecture.

Chapter 4 discusses our methodology using critical path analysis for detecting bottlenecks in early EDGE designs and presents a complete bottleneck analysis of the TFlex microarchitecture. This analysis identifies the major bottlenecks in this architecture; motivating the optimizations proposed for T3 in the rest of this dissertation.

Chapter 5 discusses the instruction mapping used by TRIPS and TFlex and proposes a new instruction mapping for T3 called *deep mapping*. This mapping significantly reduces the delay and energy associated with intra-block communication across the network, which is the number one bottleneck in

TFlex based on our initial critical path analysis presented in Chapter 4.

Chapter 6 proposes an optimization for T3 called *selective register value bypassing* that alleviates the inter-core register communication bottleneck. The proposed mechanism sends values directly from each output-critical instruction in one executing core to their consumer instructions in other cores, thus bypassing shared register forwarding units.

Chapter 7 proposes an optimization for T3 called *block reissue* that addresses fetch bottleneck caused by mispredictions. By keeping track of previously fetched blocks and reissuing those blocks if needed, this feature reduces critical time between block flush and fetch after mispredictions.

Chapter 8 presents the T3 integrated next block predictor and predicate predictor. This predictor, called Iterative Path Predictor (IPP), addresses two fundamental problems associated with speculation in EDGE architectures which are their low next-block prediction accuracy and low intra-block speculation rate. To improve next block prediction accuracy and increase the speculation rate, this predictor predicts the predicate path within each block and uses it to predict the next block and speculate on the intra-block predicates.

Chapter 9 presents an overview of exposed operand broadcasts (EOBs) which are used by T3 to address the operand delivery bottleneck caused when using dataflow for high-fanout instructions. This hybrid operand delivery mechanism uses dataflow and compiler-generated light-weight broadcasts to handle low- and high-fanout operands, respectively.

Chapter 10 compares the fully integrated T3 system to previous EDGE microarchitectures (TRIPS and TFlex) that have different core composition granularities and microarchitectural features and shows that T3 improves significantly on latency, energy efficiency and scalability. The chapter also compares the performance/power flexibility of the T3 microarchitecture against several design points in the performance and power spectrum of production processors such as Intel Atom and Core 2 processors. The results show that T3 not only performs very efficiently in low-energy and high-performance regions but also can perform in a much larger performance/energy space beyond DVFS on conventional processors. For example, composing different number of cores, not only T3 can perform in both low-energy and high-performance regions, but it also can perform in between or above those regions. Chapter 11 summarizes the dissertation, discusses the future work, and concludes.

Chapter 2

Related Work

Related work falls into three main categories. The first category is the use of distributed uniprocessors or composable cores similar to TRIPS and TFlex for scaling single thread performance by merging distributed lightweight cores. The second category includes microarchitectural or ISA techniques that are used for optimizing the processor pipeline. We review some of these mechanisms that are similar to the ones employed by T3 for implementing efficient instruction fetch and mapping, predicate and branch prediction, criticality prediction and analysis, hybrid operand delivery, cross-core register bypassing and instruction reuse. The third category is the previous work on the critical path analysis from which the bottleneck detection methodology used by this dissertation is derived. The complete review of previous EDGE architectures is presented in Chapter 3.

2.1 Distributed Uniprocssors

To support workloads with differing degrees of parallelism, multi-core systems must adapt the granularity of cores to match the available number of threads [36]. One approach to this problem is to use dynamic or composable multicores that aggregate a small number of cores to form a larger core capable of exploiting concurrency at a finer granularity [39, 41]. Recent studies propose methods for aggregating both in-order [86, 94] and out-of-order cores [39, 41]. This study relies on out-of-order core aggregation as the underlying mechanism for exploiting block-level concurrency in programs. Some architectures take a dynamic approach for aggregating independent cores while others employ compiler and ISA support to achieve this goal.

2.1.1 Fully Dynamic Distributed Uniprocessors

A recent trend has been to balance ILP and TLP by adjusting the number of distributed resources allocated to a thread, by having multiple independent units collude to accelerate a single thread dynamically. This approach makes distribution of instructions more challenging because the number of participating processor elements is unknown statically and may change dynamically. In the Federation technique [86], two neighboring in-order cores, similar to Niagara/T1 [44] cores, are "federated" to create an out-of-order processor. A recent study, however, demonstrates that aggregating in-order cores, even under idealized assumptions about aggregation overheads, leads to major performance challenges [73].

Some recent work has allowed core aggregation on a set of out-of-order cores. CoreFusion [39, 93] is a technique that "fuses" multiple dual-issue out-of-order cores to form a wide-issue out-of-order core. The fused cores form a distributed instruction cache, instruction window and branch predictor, but some of the structures, such as register renaming, are physically shared, which limits the aggregate issue width to eight. When fused, each core uses its private i-cache and branch predictor to fetch instructions and predict branches. The information about branch prediction decisions needs to be transferred to a central unit called the fetch management unit to arrange a consistent sequence of executing instructions. Fetched instructions are sent to another centralized unit for register renaming and finally to their executing cores. The use of the physically shared register renaming and fetch units causes bottlenecks and limits the aggregate issue width to eight.

To guide instruction wakeup, selection, and issue, Forwardflow [33], which is another composable system, dynamically builds an internal dataflow representation from instructions within a single thread distributed across multiple cores. To save energy, T3 uses the compiler to generate the dataflow representation. Similarly, Hybrid Dataow Graph Execution (HeDGE) [84] explicitly maintains dependences between instructions in the issue window by modifying the issue, register renaming, and wakeup logic. Using explicit consumer encoding, this architecture employs Random Access Memory (RAM) instead of Content Addressable Memory (CAM) needed for broadcast. WiDGET [93] decouples thread context management units from execution units and can adapt resources to operate in different power-performance regimes. Instead of using dedicated units for fine-grained control management, T3 exploits distributed ISA-supported block-level control mechanisms to improve scalability. Also, different from both WiDGET and CoreFusion, T3 distributes control and instruction sequencing across executing cores, thus avoiding centralized control units.

Multiscalar [80] and Thread-level Speculation [45] rely on discontinuous instruction windows by having the hardware spawn speculative compiler-selected threads on multiple cores.

2.1.2 Compiler-assisted Distributed Uniprocessors

Instead of resolving cross-core data/control dependences dynamically, some approaches take advantage of compiler support to extract instruction dependencies statically. Instruction Level Distributed Processing [42, 43] supports hierarchical register files consisting of many general purpose registers and a few accumulator registers. The hardware steers each compiler-detected strand of instructions to a processing element and its accumulator. The inter-strand dependencies are handled through the general purpose registers.

Distributed dataflow-like architectures, including Explicit Dataflow Graph Execution (EDGE) architectures can also support a varying number of dynamic elements assigned to a single thread. TRIPS uses the compiler to form predicated *blocks* of dataflow instructions and to place each instruction on a 16-ALU grid, where they are issued dynamically [75]. TFlex is a second generation EDGE design that supports dynamic core aggregation [41], and is the baseline distributed substrate used in this dissertation.

2.2 Efficiency Optimizations

This section discusses the prior work on the optimization mechanisms used by T3 for energy efficiency. These optimizations include instruction mapping, instruction communication and operand delivery mechanisms, cross-core register value bypassing, instruction reuse and path and predicate prediction.

2.2.1 Instruction Mapping

Some architectures, such as VLIW architectures and RAW, rely heavily on the compiler to map instructions to a distributed substrate. For example, the RAW compiler schedules instructions in time to exploit concurrency, and places instructions on a physical substrate [92]. The Voltron architecture [94] combines multiple in-order VLIW cores into a wide-issue VLIW core. This statically exposed architecture relies on the compiler to schedule VLIW instructions and extract fine-grained communicating threads.

Fully dynamic approaches only use hardware to map instructions. These methods do not take advantage of instruction dependencies extracted by the compiler. Clustered superscalar processors [8, 10, 15, 26, 72, 95] rely on the hardware to steer instructions dynamically to different clusters based on instruction dependencies. Complexity-Effective Superscalar Processors [57] steer the dependent instructions into separate FIFO buffers dynamically and only send the result tags to the heads of the FIFO buffers. The ISA for Instruction Level Distributed Processing [42, 43] supports hierarchical register files consisting of many general purpose registers and a few accumulator registers.

The instruction stream is divided into short strands of dependent chains. The instructions in each strand are steered into a processing element associated with the accumulator accessed by those instructions. While the instructions in each cluster are linked by the the accumulator, the inter-strand dependencies are passed through the general purpose registers. To simplify the hardware, this dissertation relies on the compiler to specify instruction dependencies and concurrency, rather than discovering them at runtime.

The runtime mapping approach presented in Chapter 5, which can use static information, is most similar to approaches in which the hardware maps coarse chunks of work to distributed units, often with compiler support. The compiler for Multicluster processors partitions instructions between clusters during register allocation to minimize remote register accesses [27, 95]. Instructions in each cluster are scheduled dynamically by the hardware. In Multiscalar [80] and Thread-Level Speculation [45], the hardware automatically spawns speculative threads, selected by the compiler, on multiple cores. These more speculative approaches rely on discontiguous instruction windows. Wavescalar is a dataflow processor that uses static placement of instructions and dynamic issue on a hierarchical substrate [85].

2.2.2 Instruction Communication Mechanisms

As operand delivery and instruction communication mechanisms is the focal points of chapters 6 and 9 in this dissertation, this subsection discusses different instruction communication mechanisms, and how different architectures employ them to handle different types of dependences. This section also discusses different optimization methods applied to each communication mechanism in the recent literature.

2.2.2.1 Instruction Communication via Registers

Registers are fast, temporary storage units for data. In superscalar machines, registers are used for handling long dependences. In other words, if the consumer instruction is not present in the instruction window when the producer instruction produces its output, the consumer will read the value of the output of the producer from a register during the dispatch phase.

Power consumption and access delay are fundamental problems when using large register files. Therefore, many studies suggest different optimization methods to improve the register access time or power consumption. A register cache mechanism is proposed in [11] to reduce the length of the critical loops in the pipeline of superscalar processors by reducing the register access time. Using multiple-banked [9, 22] register files is another technique to reduce register file access time and energy. Distributing physical registers across multiple banks, these techniques attempt to reduce the number of ports and access time per bank. Exploiting dataflow blocks, T3 relies on registers only for inter-block communication. Additionally, it uses direct register bypassing to reduce long register latencies due to distributed register forwarding logic.

2.2.2.2 Broadcast Bypass Network

Superscalar processors use registers for handling long dependences and a combination of register renaming and broadcast bypass networks for handling short dependences. Bypass network broadcasts the result of an executed instruction, along with a tag, previously assigned during the register renaming phase, to all unissued instructions in the instruction queue. Those instructions compare the tags of their operands against the broadcasted tags. If the broadcasted tag and tag of one of their operands are identical, the value of that operand will be set to the value read from the broadcast network, and the ready flag of that operand is set.

In superscalar processors with dispatch-bound register reads, instructions access the register file in the dispatch state [58]. Example of processors with dispatch-bound register reads are Pentium Pro and Power PC 604. In superscalar processors with issue-bound register reads, instructions access the register file in the issue state [58] and the operand values are not stored in the issue queue any more. Examples of processors with issue-bound register reads are Pentium 4 and Alpha 21264. In these designs, instruction queue stores and updates the status of registers corresponding to the operands of each instruction. Prior work on SPEC benchmarks shows that short dependences handled by the broadcast network constitutes about 75% of program dependences in a superscalar processor processor [31].

2.2.2.3 Dataflow Communication

Tokens or packets are used by dataflow machines for point-to-point communication among instructions. Dennis's dataflow machine [23] has an instruction memory with each instruction cell corresponding to an operation of a dataflow program. When the operands are ready, the instruction is sent through a high bandwidth switch to an operation units to execute. After instruction is executed in the operation unit, the result of the operation is sent as one or two packets (or tokens), along with the address of a the destination operand to the instruction memory. MIT TTDA dataflow machine [6] in an abstract level is similar to MIMD machines. Each PE is a dataflow processor. One I-structure (storage unit for function or threads) and one PE constitute a complete dataflow computer. Each PE runs a code block and addresses within code blocks are relative. The result token of an executed instruction is sent back to the PE, or to another PE executing the destination code block. In first generation dataflow machines [6, 23], different from the conventional von Neumann machines, data values are not permanently stored in memory or registers. Instead, data values are transmitted among instructions using tokens allowing for massively parallel the execution. However, these machine run programs written in special dataflow languages, which are not popular. In addition, there are some implementation problems that were never overcome in these machines such as difficulties in broadcasting tokens when there are several consumers and encoding all the dependences in the program,

Wavescalar [85] uses a fetch-less instruction set with an executable L1 instruction cache with L2 data caches. Each instruction in the memory contains all of the architectural states of that instruction. Each instruction can encode any number of targets. To prevent code bloat, there is only one copy of an instruction in the system. To distinguish between different instances of one instruction, the hardware uses a field called *wave*. The generation and maintenance of the waves is handled using special instructions inserted in the code by the compiler. In addition to complexity of the required executed memory, high-fanout instructions is another problem in the Wavesalar architecture.

TRIPS [14] supports a very large instruction window using a hybrid dataflow and atomic block execution model. In this processor, the instruction window holds several blocks of instructions running in parallel speculatively. These instruction blocks communicate through the memory and registers. Inside each block, however, instructions execute in a dataflow order, thus directly communicating to each other. In this ISA, each instruction encodes up to two target instructions in the same block using their offsets from the beginning of the block. If an instruction has more than two targets, the EDGE compiler [78] uses *move* instructions to generate a fanout tree to deliver the output to its targets. Although this approach fixes the high-fanout instruction encoding problem, the inserted *move* instructions incurs a performance penalty in terms execution latency and code size.

2.2.2.4 Hybrid Instruction Communication

Due to high amount of energy consumed during tag matching, the broadcast bypass networks are a major source of high power consumption in the instruction queue [58]. Many studies attempt to reduce the power dissipated during the tag matching and wake up phases in the instruction queue.

Several approaches [16, 17, 38, 62, 63] have proposed hybrid schemes which dynamically combine broadcasts and direct dataflow to reduce the energy consumed by the operand bypass. These dynamic hybrid schemes use hardware to detect instruction dependences and dynamically select the right communication mechanism for each instruction. Gonzalez et al. [16, 17] observe that many instructions only have small number of consumer instructions in the instruction window. Based on this observation, they propose a power-efficient issue logic design for superscalar processors. The approach implements a table called N-use table, which is indexed by physical register, to store the first N consumer instructions of each physical register. If a physical register has more than N consumers in the table, the next consumer instruction is put into a small out-of-order instruction queue called I-buffer, on which broadcast is performed. The instructions stored in the N-use table will get the operand through point-to-point communication, when the corresponding physical register is available. The ones in the I-buffer will get their operand through broadcast. The ratio between point-to-point and broadcast can be adjusted by changing the value of N. This approach eliminates most of the broadcasts and tag matchings. However, the N-use table is a complex structure. Multiple copies of an instruction in the N-use table need to maintain circular pointers to each other. These pointers need to be updated when the corresponding physical register is available. For an M-issue processor, it require 2*N*M read ports and N*M writing ports, which makes it unpractical for implementation on wide-issue superscalar processors.

Huang et al. [38] propose a full hardware pointer-based approach to eliminate the broadcasts and the tag-matchings, which detects the one-consumer instructions dynamically and performs point-to-point communication from them. Any instruction targeting more than one instruction has to broadcast. During dispatch, a consumer instruction updates a pointer to itself in the Instruction Queue entry associated with its producer instruction. This pointer value is used during the issue of the producer instruction to directly send the result to the consumer. This approach avoids using the complex N-use table, in stead, only performs point-to-point communication from the one-consumer instructions. However, as we show in the analytical model and results section, broadcast from any instructions with more than one consumer can not the reach the optimal point in terms of minimizing the power consumption. In this sense, T3 EOB hybrid instruction communication model presented in Chapter 9 is a generalization of Huang's model, and demonstrates that this generalization provided enhanced benefits.

Different from dynamic hybrid models, the architecturally exposed operand broadcasts (EOBs) discussed in Chapter 9 for T3 rely on the ISA to be conveyed into the microarchtecture. The involvement of the ISA provides some opportunities for the compiler while causing some challenges at the same time. The main role of the compiler is to pick the right mixture of the dataflow and EOBs such that the total energy consumed by the *move* trees and the EOBs becomes as small as possible. In addition, this mixture should guarantee an operand delivery delay close to the one achieved using the fastest operand delivery method (i.e. the EOB network).

2.2.3 Distributed Register or Memory Bypassing

Moshovos et al. propose memory bypassing and cloaking algorithms [55] to reduce memory delay in superscalar processors. These mechanisms identify dependent loads and stores early in the pipeline and speculatively bypass the store values to dependent loads prior to address calculation and disambiguation. Selective critical value bypassing proposed in Chapter 6 essentially is similar to the memory bypassing. Selective critical value bypassing, however, is designed for register value bypassing between blocks of a distributed instructions. Therefore, it is not speculative and does not need address calculation or disambiguation. Moreover, it is only applied to critical communication edges.

Krishnan and Torrellas [46] propose a hardware-based cross-core register communication in thread level speculation (TLS) systems using a *synchronizing scoreboard* and a shared bus. Restricting register bypassing to immediate successor blocks, our selective critical value bypassing proposed in Chapter 6 does not incur any of these overheads. To further reduce the overhead, it also performs cross-core value bypassing only for the predicted critical register output values.

2.2.4 Instruction Reuse

Trace processors exploit control independence by reusing control-independent traces in the window following misprediction events. The trace generation hardware implements complex algorithms for detecting fine-grain, intra-trace control-independence and coarse-grain, inter-trace global re-convergent points [54, 70, 71]. Taking advantage of the compiler-generated predicated blocks, our block reissue mechanism for T3 proposed in Chapter 7 does not use these hardware components. Moreover, each core only maintains the availability status of its associated blocks, which amortizes the bookkeeping overhead across a large number of instructions. Sankaralingam [74] et al. propose *instruction revitalization* for TRIPS in which, the compiler adds a setup block to the beginning of each loop to dynamically initiate reissuing of the loop body. The T3 block reissue method proposed in Chapter 6 leverages the same concept of block revitalization, but it is not limited to loops and is fully dynamic so no setup code is added statically.

There have also been proposals for reusing control independent instructions [5, 18, 37, 79]. Most of these proposals use complicated checkpointing mechanisms to find control/data independent instructions. This dissertation proposes a coarse-grain reissue mechanism in which full blocks of a distributed large instruction window get reissued. As a result, this mechanism amortizes the bookkeeping overhead over a large number of instructions. Additionally, this mechanisms eliminate a huge portion of energy-hungry accesses to instruction caches while reducing the effect of fetch bottleneck.

2.2.5 Path and Predicate Prediction

Previous approaches investigate predicate prediction schemes [7, 19, 51, 60, 61] for superscalar designs. To preserve the benefit of predication on hard-to-predict branches, these approaches use a restricted version of selective predicate prediction based on the estimated confidence of prediction. Chuang et al. [19] propose predicate prediction for out-of-order processors to alleviate the problem of multiple register definitions along the if-converted control paths. They reverse if-conversions by predicting the predicates, which reduces the predication penalty. To preserve the benefit of predication, this method utilizes a replay mechanism that makes the predicate misprediction penalty less than the branch misprediction. This work is extended through unifying the branch and predicate predictor to recover the correlation information loss [61]. The iterative path predictor (IPP) proposed in Chapter 8 for T3 relies on fully distributed protocols and so does not use any central integrated predictor.

A multi-level distributed branch prediction model has been used by Multiscalar [40]. Multiscalar performs two levels of branch prediction: (1) To find the next task, a central inter-task exit predictor predicts which of the four exits of a the current task will be taken. (2) Within each task, an intra-task traditional taken/not-taken predictor predicts the outcome of the branch instructions in the task. The intra-task and inter-tasks predictors operate independently. Relying on block-level distributed protocols, the T3 iterative path predictor unifies branch and predicate path predictors while exploiting the prediction results in a fully-distributed fashion. This leads to maximizing both fetch and speculation across distributed cores.

2.3 Criticality Analysis

Early work on critical path analysis generally focus on predicting critical loads instructions and prioritizing them over the non-critical ones across the cache hierarchy [30, 81, 82]. These methods only focus on predicting or evaluating critical load instructions and cannot be easily extended to other instructions or micro-architectural resources.

Fields et al. [28] and Tune et al. [91] propose a general profile-driven model for estimating program's critical path using a dependence graph. In this graph, nodes represent micro-architectural events and links between the nodes represent dependences between events. These dependences must comply with the *dependence constraints* dictated by program data dependences and the micro-architectural restrictions of the target processor. Several prior studies have shown that such a simulation-based critical path analysis is more effective for a detailed performance analysis than conventional simulation-based techniques and hardware performance-monitoring techniques [29, 90, 91]. Conventional techniques usually provide coarse-grained statistics, such as the number of cache misses and branch mispredictions. These statistics although useful are insufficient to find the interactions between components and detect system bottlenecks.

In addition to simulation-based criticality analysis, Fields et al. [28] propose a state-of-the-art criticality predictor for superscalar processors. In this design, the processor detects micro-architectural events and sends the information about each event type and associated PC to the predictor as training data. The predictor uses a forward token passing algorithm to detect long lasting chains of instructions and predict them as critical instructions. The communication criticality predictor proposed in Chapter 6 for T3 uses a simple majority vote algorithm without employing any complex token passing hardware.

Nagarajan et al. [56] extend the simulation-based critical path analysis for performance analysis of the TRIPS processor [75]. The TRIPS execution model treats large blocks of instructions as atomic units for fetch, execution, and commit. TRIPS also support a distributed microarchitecture in which numerous computation tiles communicate across a routed network. The critical path tool implements the new micro-architectural events and dependence constraints introduced by TRIPS ISA and micro-architectural features [56]. The tool proposed for bottleneck analysis in Chapter 4 for detecting TFlex bottlenecks is an extension of that work [56]. We customize this tool for the TFlex composable multicores [41]. This tool supports various configurations of this architecture. Additionally, the tool supports different levels of granularity for presenting and analyzing the criticality information, which simplifies the detection of the system bottlenecks. Finally, it supports special operation modes to evaluate the effect of speculation on performance in different configurations.

Chapter 3

Background

3.1 EDGE ISAs

Explicit Data Graph Execution (EDGE) ISAs [75] were designed with the goals of high single-thread performance, ability to run on a distributed, tiled execution substrate, and good energy efficiency. An EDGE compiler converts program code into single-entry, multiple-exit predicated blocks. The two main features of an EDGE ISA are block-atomic execution [53] and direct instruction communication within a block. Instructions in each block use dataflow encoding in which each instruction directly encodes its destination instructions. Using predication, all intra-block branches are converted to dataflow instructions. Therefore, within a block, all dependences are direct data dependences. An EDGE ISA uses architectural registers and memory for inter-block communication.

This hybrid dataflow execution model supports efficient out-of-order execution, using conceptually less energy to construct the dependence graphs, but still supports conventional languages and sequential memory semantics. In an EDGE ISA, each block is logically fetched, executed, and committed as a single entity. This block atomic execution model amortizes the bookkeeping overheads across a large numbers of instructions and reduces branch predictions and register accesses. Additionally, it reduces the frequency of control decisions, providing the latency tolerance needed to make distributed execution, across multiple tiles or cores, practical.

3.2 The TRIPS Tiled Architecture

TRIPS was the first-generation microarchitecture to use an EDGE ISA. The TRIPS ISA supported fixed-size EDGE blocks of up to 128 instructions, with 32 loads or stores per block. Instructions could have one or two dataflow targets, so instructions with more than two consumers in a block employed `move` instructions, inserted by the compiler to fan operands out to multiple targets.

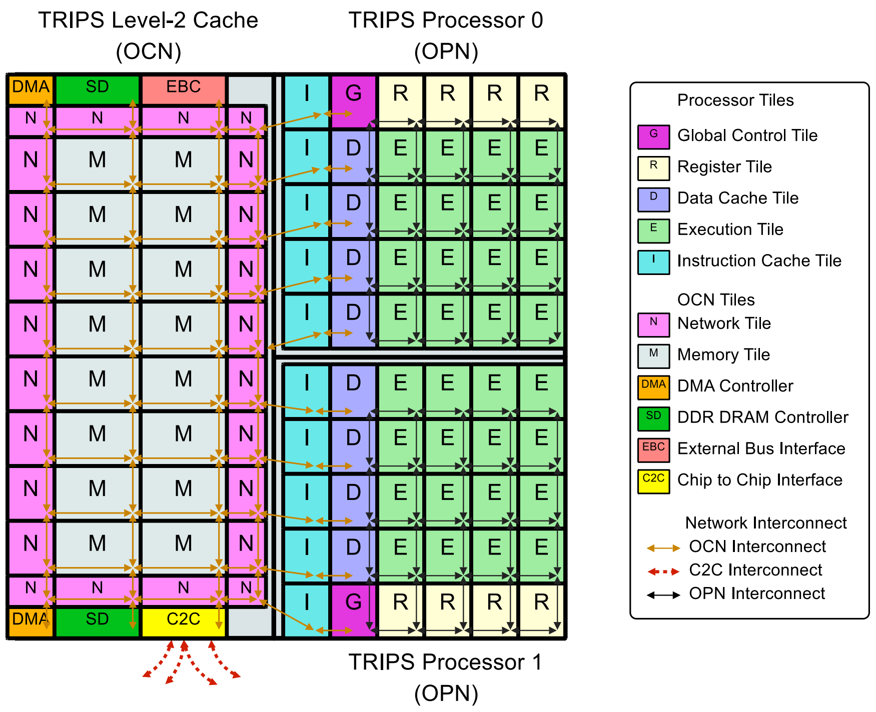
To achieve fully distributed execution, the TRIPS microarchitecture used no global wires, but was organized as a set of replicated tiles communicating on routed networks. Figure 3.1(a) shows a TRIPS tile-level block diagram. Each TRIPS processor used five types of tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The TRIPS microarchitecture could simultaneously execute up to eight atomic blocks (one non-speculative, seven speculative) for an aggregate instruction window size of 1024 instructions. The GT tile was in charge of maintaining the control order of the in-flight blocks, performing next-block prediction and initiating block allocation and block commit/flush.

The TRIPS design had a number of serious performance bottlenecks [32].

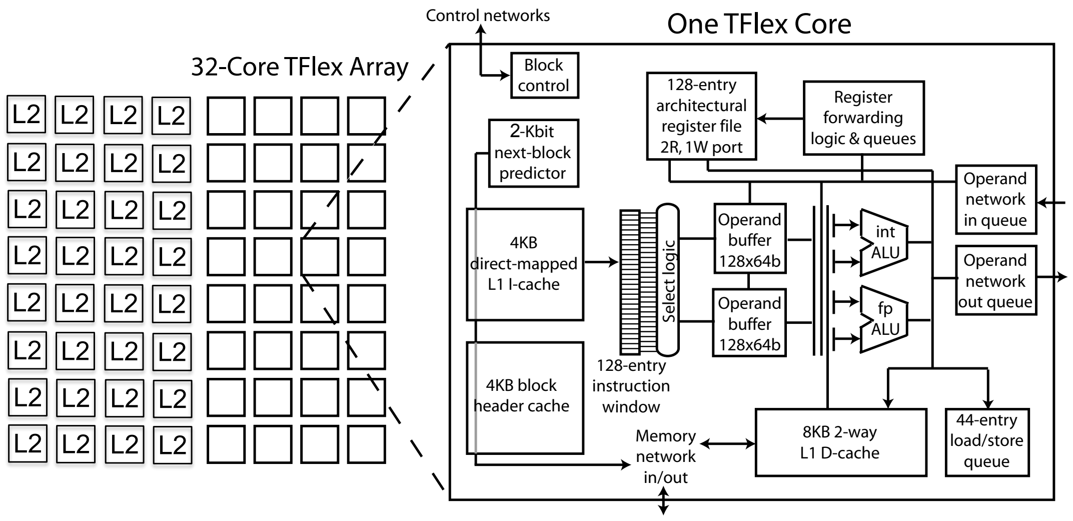
Misprediction flushes were particularly expensive, because the TRIPS next-block predictor had low accuracy compared to modern predictors, and the refill time for such a large window was significant. Since each instruction blocks was distributed among the 16 ETs, intra-block operand communication was expensive, both in terms of energy and latency on the critical path. The predicates used for intra-block control also caused performance losses, as they were evaluated in the execution stage, but would have been predicted as branches in a conventional superscalar design. Finally, the RTs and DTs distributed around the edges of the ET array limited register and primary memory bandwidth, and forced some instructions to have long routing paths to access them.

3.3 The TFlex Composable Microarchitecture

TFlex was a second-generation EDGE microarchitecture [41], which used the TRIPS ISA but improved upon the original TRIPS microarchitecture. TFlex distributed the memory system and control, making each tile a fully functional EDGE core, but permitting a dynamically determined number of tiles to cooperate on executing a single thread. Thus, TFlex is a dynamic multicore design, similar in spirit to CoreFusion [39]. The ability to run a thread on a varied number of cores, from one to 32, was a major improvement over TRIPS, which had a fixed execution granularity. That fixed granularity made it unable to adapt the processing resources as the workload mix, application parallelism, or energy efficiency requirements changed. The important



(a) TRIPS block diagram [32]



(b) TFlex core array and the components in a single TFlex core [41]

Figure 3.1: TRIPS and TFlex block diagrams

Table 3.1: TRIPS and TFlex microarchitecture comparison in 45nm.

Structures	TRIPS (16-issue)		TFlex-1 (2-issue)	
	Size	Area (mm^2)	Size	Area (mm^2)
Fetch (B.Pred., I-cache)	64-Kbit Block Predictor in [64] 80KB I-cache	1.45	16-Kbit Block Predictor in [41] 8KB I-cache	0.73
Reg. Files	512 entries	0.57	128 entries	0.17
Exec. resources (issue window, ALUs)	1024 entries SRAM-based issue window INT(16),FP(16)	7.47	128-entry SRAM-based issue window 2-INT ALU, 1-FP ALU	0.73
L1 D-cache (D-cache, LSQ)	32KB D-cache 1K-entry LSQ	6.35	8KB D-cache 44-entry LSQ	0.68
Routers	OPN [32]	2.09	Dual OPN [41]	0.19
L2 Caches	4-MB NUCA Cache	–	4-MB NUCA Cache	–
Total		17.93		2.50

new features supported by TFlex are as following:

Distributed Register and Memory Systems: Unlike TRIPS, which distributed the DTs, ITs, and RTs along the edges of the execution array, limiting bandwidth and scalability, the TFlex microarchitecture distributes the register, data caches, and instruction caches across all participating cores as interleaved banks. This change required special functionality in the load/store queues, using flow control to NACK loads or stores that would overflow an LSQ bank in one of the participating cores.

Distributed Control: TRIPS maintained the processor control and sequence of program execution in a single tile (the GT in Figure 3.1(a)). TFlex distributes the control responsibilities across all participating cores. This microarchitecture employs distributed protocols to implement next-block prediction, fetch, commit, and misprediction recovery using no centralized logic, enabling the architecture to scale to 32 participating cores per thread.

Figure 3.1(b) illustrates the microarchitectural components of a TFlex core [41]. Table 3.1 compares the area and size of different microarchitectural components of TRIPS and TFlex in the 45nm technology. Each TFlex core has the minimum resources required for running a single block, including a 128-entry RAM-based instruction queue, an L1 data cache bank, a register file, a branch prediction table, and an instruction (block) cache bank.

When N cores are merged, they can run N blocks simultaneously, of which one block is non-speculative and the rest are speculative. Each block is mapped to the instruction queue of one core, thus all instructions inside that block execute and communicate within the core [67]. In the merged mode, the register banks, instruction cache banks, and data cache banks of the cores are shared among the cores and are address interleaved. For example, each core contains a data cache and the low-order bits of each memory address determines the core containing the cache bank associated with that memory address [41]. In the merged mode, a register forwarding unit and a load store queue unit on each core are in charge of holding speculative register and memory values produced by the running blocks. Additionally, the register forwarding unit resolves dependences and forwards register values between blocks. Therefore, a register value produced by a block needs to be first sent to its *home* core so that its consuming blocks can be identified and it can get forwarded to the cores running those blocks. Consequently, there is no centralized renaming mechanism for inter-block register communication. Additionally, distributed protocols implement next block prediction, fetch, execute, commit, and misprediction recovery using no centralized logic, which makes this architecture very scalable. Also, the block-level prediction, fetch, commit, misprediction recovery overheads are amortized across all instructions in each block.

Similar to TRIPS, the original TFlex design distributed the instructions from each in-flight block among all participating cores, increasing operand communication latency. TFlex also had many of the same problems as the TRIPS architecture: the software fanout trees, poor next-block prediction ac-

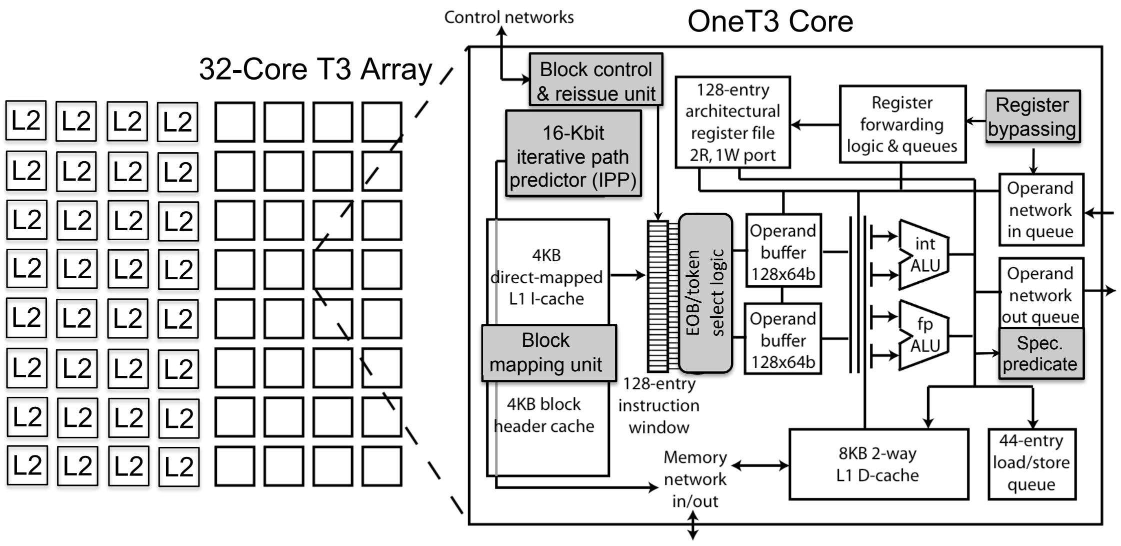


Figure 3.2: T3 Block Diagram.

curacy, predicates not being predicted, expensive refills after pipeline flushes, and large, fixed-size blocks that could cause significant instruction cache pressure. The T3 microarchitecture was designed to address these limitations of TFlex.

3.4 T3 Microarchitectural Features

The T3 microarchitecture currently uses the TRIPS ISA with one change in the instruction encodings and semantics to support Exposed Operand Broadcasts. In addition, it has five new microarchitectural features, which affect its datapaths and design significantly. These features, which are Deep Block Mapping, Critical Register Bypass, and Block Reissue, Exposed Operand

Broadcasts and the Iterative Path Predictor, mitigate all of the major performance bottlenecks identified by the analysis of the TRIPS hardware [32] and the TFlex design[41]. Figure 3.2 shows the T3 microarchitecture block diagram; the new microarchitectural components are shaded. These five additions achieve the following:

Reduced operand network traffic: By spreading blocks across all participating execution tiles or cores, both TRIPS and TFlex suffer high intra-block communication. T3 employs a mechanism called Deep Block Mapping [67] that maps each block to the instruction queue of one core, permitting all instructions to execute and communicate within the core [67]. For single-issue cores (such as the TRIPS ETs), the original block mapping shows higher performance than the Deep Mapping, but T3, like TFlex, uses dual-issue core for which Deep Mapping is most performant.

Critical inter-block value bypassing: To reduce inter-block register communication delay, T3 employs an optimization mechanism called *selective register value bypassing* [68] that bypasses remote register forwarding units by sending register values predicted to be critical directly from producing to consuming cores.

Reduced pipeline flush penalties: To alleviate branch flush penalties, T3 employs a method called *block reissue* [68]. This technique permits previously executed instances of a block to be reissued while they are still in the instruction queue, even if they have been flushed. This method saves both pipeline fill latency and energy by reducing accesses to the shared instruction cache banks.

Wide-operand fanout reduction: TRIPS and TFlex rely solely on the intra-block dataflow mechanism to communicate intra-block operands. The resulting `move` instruction trees add considerable overhead, both in time and energy. Alternatively, T3 employs architecturally exposed operand broadcast operations (EOB). The extended ISA combines dataflow operand delivery and compiler assigned EOBs to handle low- and high-fanout operands each in a power-efficient manner. EOBs nearly eliminate this overhead and results in a major energy saving.

Branch and predication overhead reduction: T3 employs a new predictor design called an Iterative Path Predictor, which unifies branch target and predicate prediction while providing high accuracy for each. This predictor generates four bits of a predicted path within the block per cycle, quickly obtaining the predicated control path through the block, to generate the predicted next-block target. The bits from this path are then used to predict the predicates within the block.

3.5 Summary

Table 3.2 compares features of the TRIPS, TFlex and T3 architectures. TRIPS and TFlex both employ TRIPS ISA and T3, which is evaluated in this dissertation, uses a slightly modified TRIPS ISA .

T3 can be considered a major improvement on previous EDGE archi-

Table 3.2: Features supported by different EDGE architectures.

	ISA	TRIPS			E2
Category	Microarchitecture	TRIPS	TFlex	T3	E2
Distributing instructions	Distributed, tiled EDGE microarchitecture	X	X	X	X
	Fully distributed register files and L1 caches		X	X	X
	Variable core granularity (Dynamic Multicore Capability)		X	X	X
	Reduced operand network traffic (Deep Mapping)		X[1]	X	X
Operand delivery	Inter-block latencies reduced (Critical Register Bypassing)			X	X
	Wide operand fanout reduced (EOB)			X	X
Distributed Speculation	Branch flush penalties reduced (Block Reissue)			X	X
	Predicate resolution bottleneck reduced (IPP)			X	X
	Improved next-block target prediction (IPP)			X	X
Instruction packing	Variable-sized blocks				X
	SIMD & vector instructions				X

tectures in the following aspects of the microarchitecture and ISA:

Instruction distribution: Taking advantage of block-based dataflow execution, TRIPS and TFlex distribute instructions across distributed tiles with low control synchronization overhead to maintain a distributed sequence of instructions. In these architectures, the compiler distributes instructions in each block across all cores. However, using this mapping, the delay of cross-core dataflow communication among instructions is high. By mapping all instructions in each block to only one core (which is called deep mapping in this dissertation), T3 eliminates this bottleneck while saving energy consumed over the on-chip network.

Operand delivery: TRIPS and TFlex exploit dataflow and registers for intra- and inter-block operand delivery, respectively. Dataflow is inefficient for handling high-fanout operands and inter-block register forwarding latency can be high when using deep mapping. To address intra-block communication efficiently, T3 combines dataflow with light-weight architecturally exposed broadcasts; thus minimizing overall operand delivery energy within each core. To speedup inter-block register communication, T3 bypasses critical late registers directly from source cores to destination cores; thus shortening critical path latency.

Speculation: To generate large number of in-flight instructions, EDGE architectures use a next block predictor. As each EDGE block is multi exit, the

[1]The original TFlex microachitecture [41] did not use this later-proposed optimization [67].

prediction accuracy of early EDGE architectures is not as high as conventional taken/not taken predictors. To improve this accuracy, T3 exploits a predictor that quickly obtains the predicated control path through the block, and then uses this predicted path to generate the predicted next-block target. Another issue with early EDGE designs is the fact that intra-block predicated branches are evaluated in the execution stage instead of being predicted similar to the way branches are handled in conventional architectures. T3 uses the predicted predicate path to speculatively execute these predicates.

For completeness, Table 3.2 also shows the capabilities that will be enabled by the E2 ISA and compiler, briefly discussed in Chapter 11. This architecture improves EDGE design in another aspect which is instruction packing. Enabling variable block sizes and SIMD/vector instructions, this architecture significantly reduces the execution energy overheads.

Chapter 4

Bottleneck Analysis for EDGE Architectures

4.1 Introduction

Bottleneck analysis and removal of the future distributed systems can be very challenging because of their distributed nature and use of different micro-architectural components. For example, choosing the right microarchitectural optimization to improve system efficiency depends on the detailed scenarios causing the bottlenecks in the system at a given step of optimization.

This chapter proposes a method based on the profile-driven critical path analysis [28, 90] to perform bottleneck analysis and reduction for such distributed systems in a very systematic way. At a system level, this analysis reports the distribution of the critical cycles between different distributed components in the system. Considering how this distribution changes between different configurations, the system-level breakdown can detect different performance and scalability bottlenecks. At a component level, the analysis reports the critical cycles of each system component according to detailed micro-architectural event sequence leading to critical cycles. Combining the two levels of analysis, the method presents a systematic way of both detecting and reducing bottlenecks in the system iteratively. At each step of the optimization process, the system-level breakdown reveals both performance bottlenecks in each system configuration and scalability bottlenecks across a range of configurations. The component-level analysis of each bottleneck then identifies the micro-architectural cause of that bottleneck. The right optimization mechanism can be then selected for reducing the most dominant bottleneck based on the detected detailed cause of that bottleneck. Then the optimization process repeats for the one-step optimized system. This methodology for detecting bottlenecks is general and can be employed to study inefficiencies in future distributed systems in a methodical and automated way.

This chapter uses the proposed methodology to systematically optimize the baseline TFlex EDGE dynamic uniprocessor. Using these results, the chapter quantitatively discusses the effect of micro-architectural components and speculation in different system configurations. The results show how the system criticality pattern changes across different configurations in the baseline system. Some performance bottlenecks grow from light configurations with small numbers of cores to heavy configurations with high numbers of cores. These bottlenecks are identified as scalability bottlenecks. The rest of the dissertation then discusses several optimization steps for alleviating the bottlenecks in this architecture using the proposed methodology. When guided by this systematic method, the microarchitectural optimization mechanisms can optimize the system very effectively. This analysis and iterative approach for applying optimizations detects and reduces five inefficiencies in the studied distributed uniprocessor. Our evaluation shows that this analysis methodology is very effective in understanding the interactions between system components. It is also very useful in detecting and analyzing performance scalability bottlenecks and applying optimizations to remove those bottlenecks in an automatic and systematic manner.

4.2 Criticality-based Bottleneck Analysis

Using simulation-based critical path analysis for detecting bottlenecks in a system is more effective and reliable for performance analysis than conventional simulation-based or profile-based techniques. The profile-based methods achieve a coarse-grain view of the system behavior which is not sufficient for analyzing the interactions between components and detecting bottlenecks [29]. Moreover, for simulation-based methods, the space of several parameters available to the designer can become too large, thus preventing a complete analysis of different components. A simulation-based critical path analysis [28] generates and processes the program dependence graph. This dependence graph is a directed acyclic graph, where nodes represent various micro-architectural events and edges represent both data or micro-architectural dependences among these events. Micro-architectural dependences are dictated by the characteristics of the micro-architectural components of the target processor such as branch predictor, fetch, issue, commit and memory units. Based on a *last-arriving rule*, in such a graph, if a dependence edge between nodes n and m is on the critical path, the value produced by n is the last value arriving at m. Therefore, starting from the last program event and back tracking along the last arriving edges, the tool can calculate the longest dependence path which is the same as the program critical path.

We modify a cycle-level TFlex simulator [41] such that it outputs a trace of the various micro-architectural events that occur during the execution of a program. Each event in this trace includes all of the data needed to build the dependence graph such as the cycle of the event occurrence, the block associated with the event, etc. To generate the program dependence graph, the critical path tool adds links between the events (nodes) according to the dependence constraints (rules) dictated by the program data dependences and micro-architectural characteristics. For a complete review of the dependence constraints and rules for EDGE and superscalar architectures, please refer to [56] and [28].

Generating the entire program dependence graph before calculating the critical path could become intractable due to high memory requirements [56]. Nagarajan et al. [56] propose an algorithm that maintains a subgraph of events for a sliding window of $r + w$ blocks, where r depends on the total available window and w is the maximum number of blocks in flight. At each step, the algorithm constructs the graph for r consecutive blocks and then performs a backward analysis on this subgraph. The partially collected estimated critical path is then conveyed to the next step in which the next execution window is processed. The algorithm continues until the end of the dependence graph is reached [56]. We employ this algorithm for calculating the critical path. The next subsection explains our bottleneck-detection methodology that utilizes this algorithm. This methodology is general and can be applied to any architecture. Additionally, this methodology can be automated to be performed in two levels of analysis.

4.2.1 System and Component-level Analyses

In a distributed system, several distributed components interact together to execute a program. For such a system, a designer needs to analyze the criticality information at different levels of granularity. Categorizing the events only according to their instruction types may not be sufficient for understanding system inefficiencies and bottlenecks. Criticality information can be used to calculate the contribution of each micro-architectural component to the critical path. Moreover, if a component consumes a large portion of critical cycles during execution, the designer should be able to find out the micro-architecutral scenario (sequence of events) leading to the states consuming those cycles. Knowing such scenarios can help the designer develop mechanisms to alleviate the detected bottlenecks.

In our framework, in addition to its type (i.e. instruction fetch or execute) and timing information, each event includes other information needed for performance analysis and bottleneck detection. Each event has a tag indicating the component in which this event took place. Summing the delays associated with all nodes with the same tag, this analysis can estimate the critical path contribution of the component associated with that tag. We call this breakdown report of critical components the *system-level* breakdown of the critical path. The following components are reported in our system-level results for TFlex:

1) **branch misprediction** and 2) **load violation**: branch and load/store dependence misprediction overhead, respectively.

3) **data misses**: the time spent on data misses.

4) **instruction execution**: the execution time spent in ALUs.

5) **network**: the time spent on value communication across the network.

6) **fetch stalls**: the time waiting for a fetch-critical block to be fetched.

7) **block commit**: the time between when a block has finished executing all its instructions and waits for the previous blocks to commit before it can commit.

8) **instruction fetch and decode**.

9) **write forward** and 10) **store forward**: the times spent in register files or load/store queues when a register or memory value is forwarded between speculative blocks, respectively.

In the system-level breakdown, each segment of the critical path is associated with one of the explained micro-architectural components. In a given system configuration, the component that consumes a large portion of critical cycles can be considered as a potential *performance bottleneck* for that particular configuration. To identify *scalability bottlenecks* across different core counts, we run multiple configurations with various numbers of merged cores and observe how the system-level distribution of critical cycles changes. If a component consumes more critical cycles as the number of core goes up, this component does not scale efficiently as more resources are added to the system. Consequently this component can be considered a scalability bottleneck. A complete analysis for detecting performance bottlenecks in different configurations and scalability bottlenecks will be presented in this chapter.

To study the bottlenecks at a finer granularity, each event has another tag indicating the state of the corresponding component that leads to that event. For each detected bottleneck, the analysis reports the breakdown of the critical cycles corresponding to different possible states of the corresponding component. We call this breakdown report of an individual component the *component-level* report of that component. Studying the component-level results of a bottlenecks can help the designer understand under what scenarios that component becomes a bottleneck. The two bottlenecks discussed later in the section are the *on-chip* network and *fetch stalls*. For the on-chip network, the communication events are categorized in the component-level analysis as following:

1) **register communication**: inter-block communication between instructions producing or consuming a register value and the register forwarding unit on the home core of that register.

2) **memory communication**: inter-block communication between load and store instructions and the load/store queues on the core containing the loaded or stored memory location.

3) **intra-block communication**: intra-block communication needed

for direct data-flow operand delivery between distributed instructions of the same block.

4) **others**: any other type of cross-core communication such as traffic caused by the distributed fetch protocol.

Fetch stalls usually become critical if the input data of a block is already computed by the previous blocks while that block has not been fetched yet. Our component-level breakdown categorizes fetch stalls into the following categorizes:

1) **full window**: When the pipeline is full, no more block can be fetched.

2) **fill up**: When the window is not full but the control sequence has not reached the block corresponding to the event.

3) **bpflush** and 4) **ldflush**: When the corresponding block is the block immediately fetched following a branch misprediction or load/store violation, respectively.

The component-level breakdown for data misses can be categorized for L1, L2 and memory accesses. This breakdown for instruction execution can be categorized based on the type of the executed instructions or the ID of the core executing each instruction. Other categorizations can be imagined for the rest of the components reported by the analysis. This framework can be easily extended to other architectures with distributed components by reassigning system-level and component-level tags in the critical path analysis of the target architecture.

4.2.2 Speculation-Aware Mode

Evaluating the speculation overhead for both superscalar and composable processors is important because it highly affects the efficiency of the whole system. Accounting for speculation overhead, which is represented by *branch misprediction* and *load violation* components in the system-level breakdown, can be misleading. The misprediction overhead associated with a misprediction event can be defined as the time between the prediction event and the time the misprediction is detected. During this period of time, the work done by the instructions on the misspeculated parts of the code is wasted. However, the instructions leading to detection of that misprediction are useful instructions and likely to be on the critical path.

To measure the effect of speculation on the critical path, we run the critical path analysis in two different modes called *speculation-aware* and *speculation-unaware* modes. In the *speculation-aware* mode, for every branch or data dependence misprediction, there is a dependence between the event initiating that prediction and the corresponding misprediction event in the program dependence graph. In other words, the misprediction overhead is measured as the amount of work between the prediction and misprediction events. In this mode, the critical path analysis treats misspeculation overhead as a virtual micro-architectural component. Although using the speculation-aware mode is useful for understanding the effect of speculation overhead on the critical path, it can hide the effect of other critical components on execution. In the *speculation-unaware* mode, there is a dependence between the instruction detecting each mispredction and that misprediction event in the program dependence graph. Therefore, the misspeculation overhead is represented as only the time between execution of the instruction detecting the misspeculation (branch misprediction or load/store violation) and the time when the misspeculation occurs.

4.3 Analysis Results

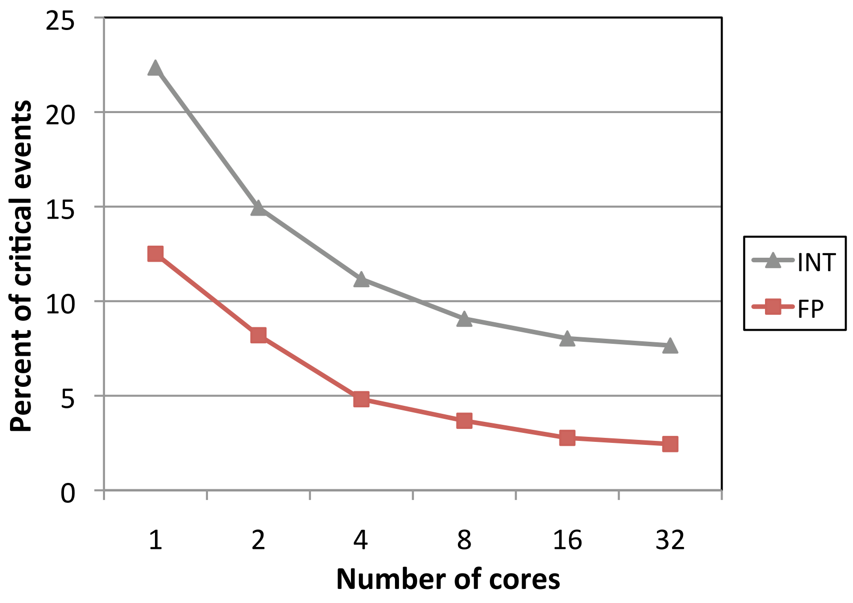
4.3.1 Methodology

This section presents a complete criticality analysis for the baseline TFlex composable multicore system at both system and component levels. We modify a cycle-level TFlex simulator [41] to add the support for generating event traces of benchmarks. We also extend the critical path tool proposed by Nagarajan et al. [56] to work with TFlex micro-architecture and to support system-level, component-level and speculation-aware modes. We use six integer SPEC2K [2] benchmarks, including *gzip, vpr, crafty, mcf, perl* and *twolf*, and six floating-point SPEC benchmarks including *wupwise, swim, applu, mesa, equake* and *ammp.* Increased simulation time when running critical path analysis prevents us from reporting the results for the other benchmarks. The critical path analysis in both system-level and component-level increases the simulation time by about 4x for most of the benchmarks. Unless otherwise specified, results presented in this section are produced using speculation-unaware analysis. Each baseline TFlex core used in this experiment is a dual-issue, out-of-order core with a 128-instruction window. Table 3.1 shows the microarchitectural parameters for each TFlex core used in this experiments.

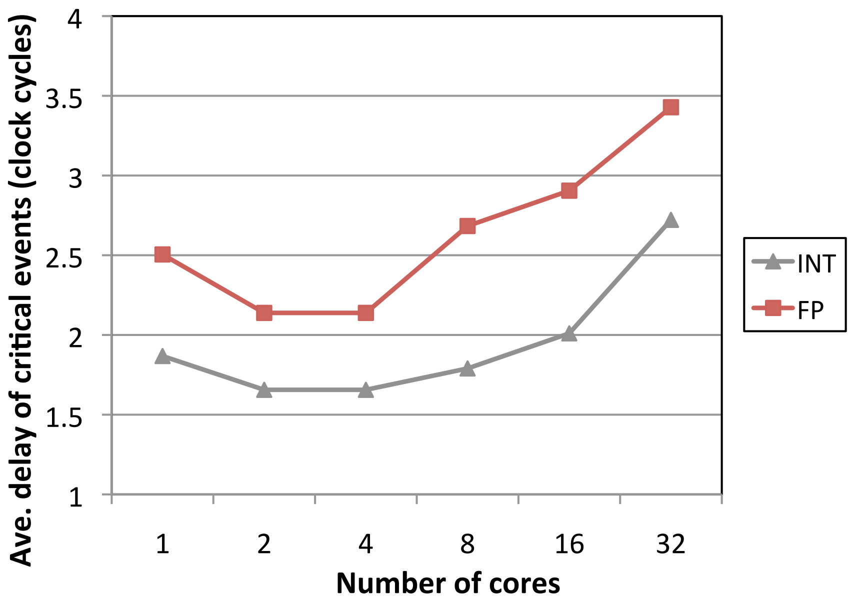
4.3.2 Critical Path Statistics

Figure 4.1 shows the percentage of critical events (critical path length) and the average delay of one critical event for INT and FP benchmarks across different system configurations. Each system configuration is associated with a fixed number of merged cores. When running on a single core, 22% and 13% of events are critical for INT and FP benchmarks, respectively. As the number of cores goes up to 32, these rates go down to 8% and 3% for INT and FP benchmarks, respectively. This reduction in critical path length is due to improved block-level parallelism resulting from having more blocks inflight. The larger percentage of critical events in INT benchmarks indicates denser control or data dependences between instructions in these benchmarks. On the other hand, FP benchmarks observe higher reduction rate of the critical event portion as more cores are merged. This indicates that FP benchmarks contain more instruction- and block-level parallelism and are able to exploit the additional resources available as more cores are added.

As shown in Figure 4.1(b), the minimum delay of critical events is achieved when merging 2 or 4 cores and then the delay starts to increase. Although the portion of critical events show constant decrease for both INT and FP benchmarks, the average delay of the events does not follow that pattern. Consequently, a constant increase in speedup may not be expected as more cores are merged. This increase in average delay can indicate that one



(a) Percent



(b) Delay

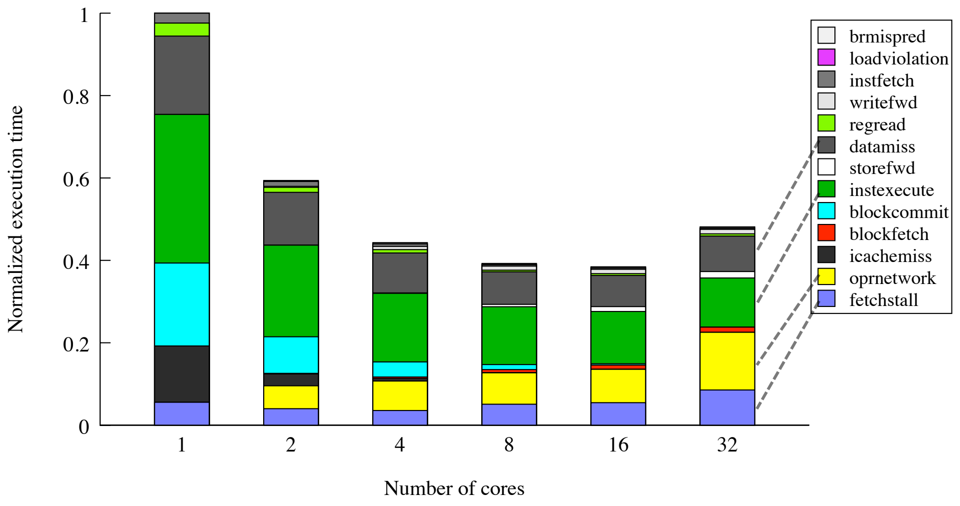
Figure 4.1: Percent and average delay (in number of clock cycles) of the critical events for SPEC INT and FP benchmarks.

or more resources are not scaled properly so causing scalability bottlenecks as the number of merged cores increases.

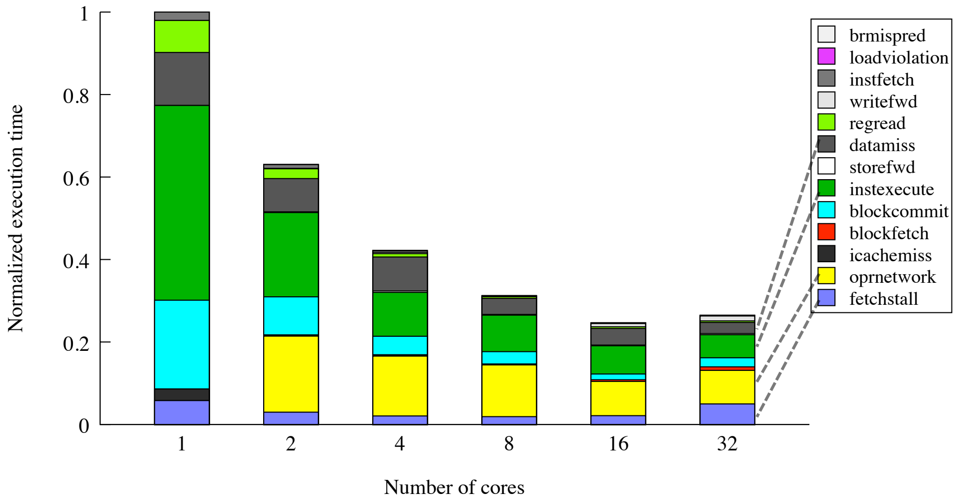
4.3.3 Scalability Bottlenecks

This subsection discusses the system bottlenecks using the proposed methodology. Figure 4.2 reports the system-level critical path breakdown for SPEC INT and FP benchmarks. Different stacked bars represent different system configurations. Each segment of the critical path is normalized against the corresponding segment length in the 1-core configuration. Therefore, the total height of the stack for each configuration represents the average execution time in that configuration normalized against the execution time in the 1-core configuration. Because there are many components in the critical path, to help the reader, the dotted lines in the figure highlight the major components in the critical path.

For INT benchmarks, when running on only one core, the dominant bottlenecks in the system are *execution bandwidth*, *block commit*, *register read* and *data misses*. As more cores are composed, execution bandwidth, block commit bandwidth and data cache and shared register capacity/bandwidth are increased. Consequently, *execution, block commit*, *register read* and *data misses* become less critical. This trend continues for the configurations with core counts smaller than eight. When using eight or more cores, *data misses, instruction execution, on-chip inter-core network* and *fetch stalls* are the major contributors of the critical path. Among these contributors, the contributions



(a) SPEC INT



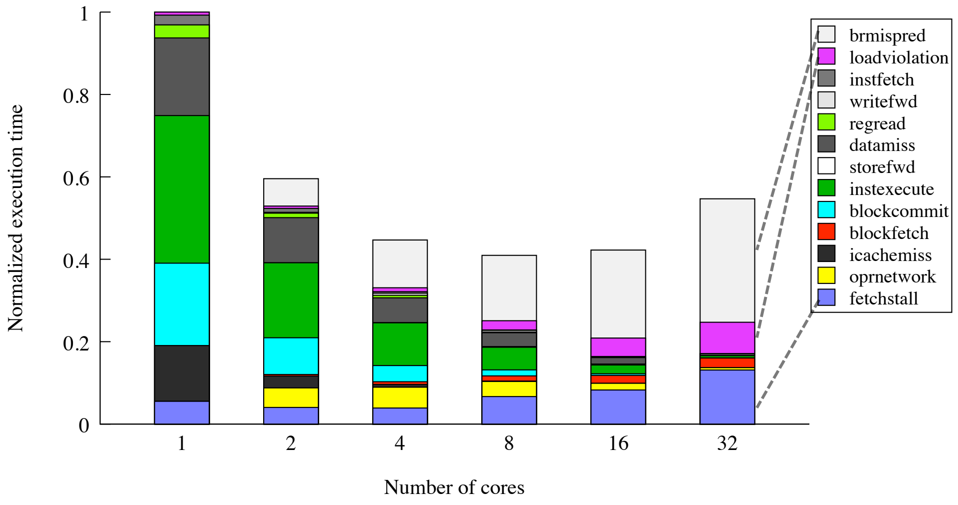
(b) SPEC FP

Figure 4.2: Critical path breakdown for different micro-architectural components.

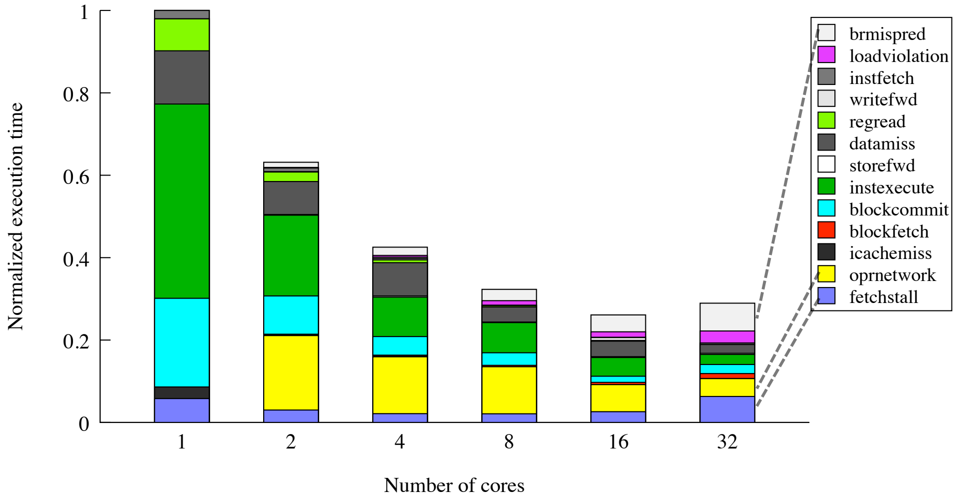
of the *on-chip network* and *fetch stalls* (the lowest two segments in each configuration shown) increase as more cores are merged. Therefore, *On-chip Network* and *Fetch* can be considered as the system's main *scalability bottlenecks*. As a result of these bottlenecks, performance stops improving and even degrades when merging 32 cores. FP benchmarks contain more parallelism and less dependences therefore, their performance scales better than INT benchmarks.

4.3.4 Speculation Bottleneck

Before we discuss component-level results for the detected scalability bottlenecks, we discuss the speculation-aware results of the system. Figure 4.3 reports the critical path breakdown for SPEC INT and FP benchmarks when considering speculation overhead as a virtual component (the two top most segments in each bar). According to these results, when merging 32 cores for the INT benchmarks, more than 60% of the critical path is composed of the instructions leading to detection of that misprediction. This part of execution does not take advantage of the parallel cores in the system because the work performed in the subsequent blocks is flushed when mispredictions occur. This overhead, however, is much smaller for FP benchmarks (30% when merging 32 cores). This can indicate a higher branch and load dependence prediction accuracy for for FP benchmarks compared to INT benchmarks when having a high number of code blocks in flight. An interesting observation in Figure 4.3 is that when only considering speculatiopn-aware results, the



(a) SPEC INT



(b) SPEC FP

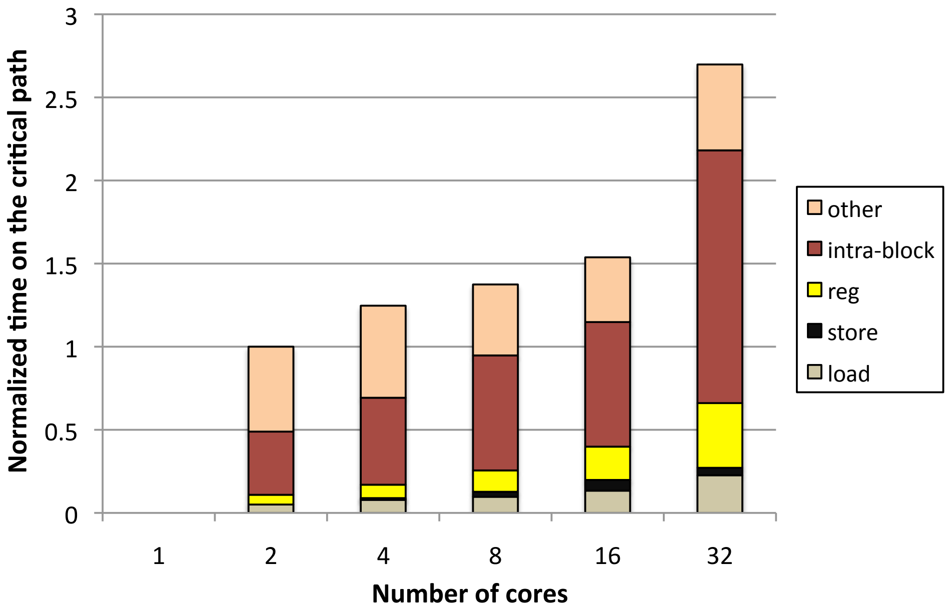
Figure 4.3: Speculation-aware critical path breakdown for different microarchitectural components.

effect of misspeculation overhead can be misleading. For example, as shown in Figure 4.3(a), speculation-aware results for INT benchmarks completely hides the effect of on-chip network communication bottleneck previously observed in Figure 4.2(a). However, the fetch stalls become more dominant when using speculation-aware results. Chapter 8 discusses Iterative Path Prediction, an integrated branch and predicate predictor designed for alleviating speculation bottlenecks in TFlex.

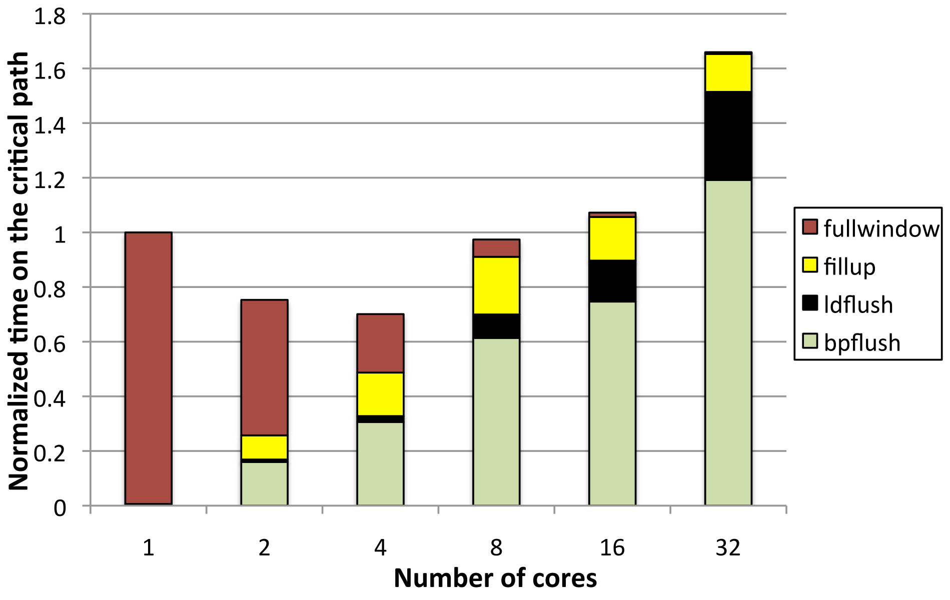
4.3.5 Communication and Fetch Bottlenecks

To further understand the sources causing the bottlenecks discussed so far, Figure 4.4 reports the component-level results for the on-chip network and fetch components for INT benchmarks. The results for FP benchmarks are similar and are not included for brevity. The on-chip network and fetch results are normalized against 2-core and 1-core results, respectively. According to Figure 4.4(a), as the number of cores increases, most of the critical cross-cores communication is caused by the operand delivery between distributed instructions belonging to the same block of the code. As the core count increases, the network distance increases and so does the average intra-block communication delay. This prevents performance scalability of the system. Chapters 5 and 6 discuss methods employed by the T3 microarchitecture for alleviating cross-core communication bottlenecks.

As shown in Figure 4.4(b), when merging a small number of cores, most of the critical fetch stalls happen because the instruction window is full most



(a) Onchip Network



(b) Fetch

Figure 4.4: Breakdown of critical cycles for system bottlenecks (Network and Fetch) for *SPEC INT* benchmarks.

of the time. Merging more cores improves the fetch bandwidth and reduces the critical fetch stalls. When using higher numbers of cores, fetch stalls on the critical path are mostly associated with the blocks immediately following misspeculation events. In these configurations, a large window of speculative instructions is constructed and consequently fetch stalls caused by misspeculation flushes are likely to end up on the critical path and become a performance bottleneck. The effect of mispredictions becomes more dominate as more cores are merged, causing another scalability bottleneck in the system. Chapter 8 discusses how the T3 microarchitecture alleviates this fetch bottleneck.

Figure 4.5 illustrates the workflow of the bottleneck detection and elimination using our analysis applied this distributed uniprocessor. The ovals labeled "System" and "Component" represent system and component level analysis used to detect bottleneck in each step of the process, respectively. Dashed lines in this figure represent bottleneck reductions and the name close to each dashed line indicate the applied optimization mechanism for reducing that bottleneck. The optimization mechanisms are discussed in the rest of this dissertation. Finally, the rounded rectangles represent bottlenecks detected in each step.

4.4 Summary

This chapter proposes a methodology based on critical path analysis to detect the scalability bottlenecks of distributed uniprocessors. This analysis is performed in different levels of granularity. First, a system-level breakdown

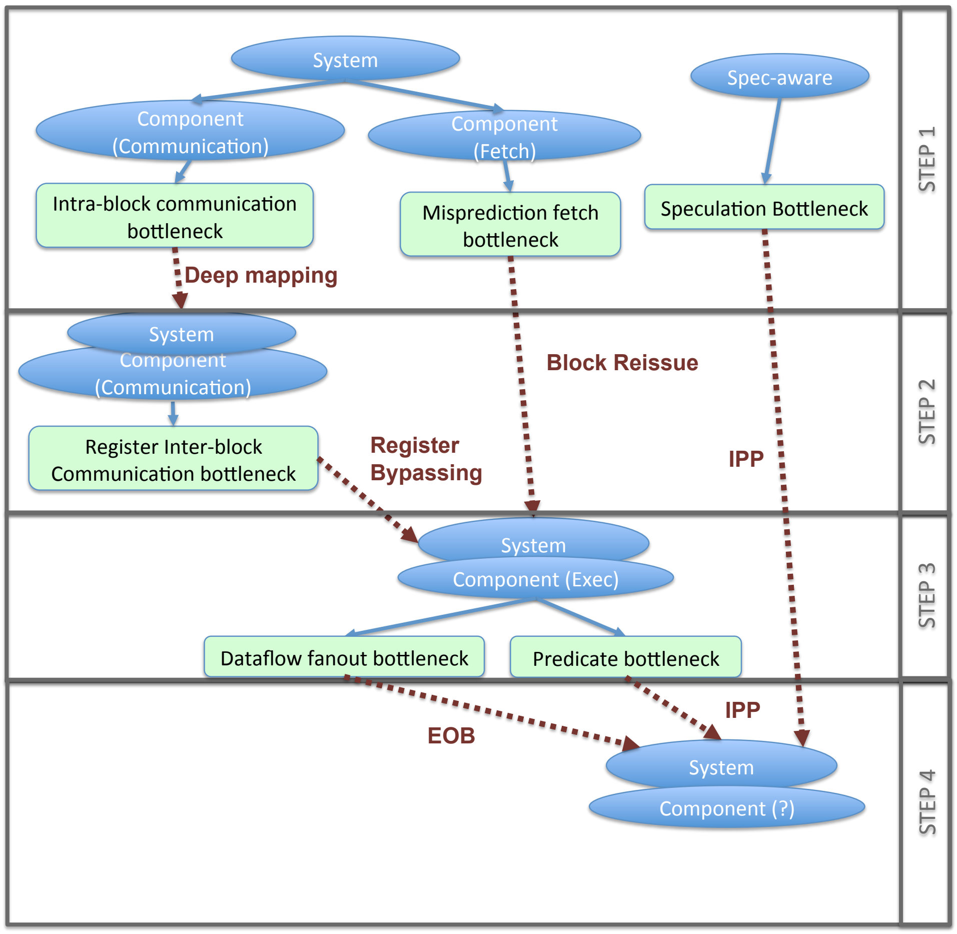


Figure 4.5: Workflow of the iterative bottleneck reduction for TFlex.

of critical cycles detects the micro-architectural components in the system that are causing scalability bottlenecks. Second, a component-level breakdown of critical cycles for each critical component identifies the conditions under which the bottleneck appear in the system. As a result, this analysis not only determines the micro-architectural bottlenecks in the system but also identifies the scenarios initiating those bottlenecks.

To effectively eliminate bottlenecks in a systematic way, this methodology can be used for iteratively applying new optimization mechanisms. At every step, the next optimization mechanism is selected based on the most dominant bottleneck detected using system- and component-level analysis. At each step, this analysis evaluates new optimization mechanisms added to the system. This analysis can be used for evaluating new optimization mechanisms added to the system. A new optimization mechanism can have positive effect on the functionality of some micro-architecutral components while underperforming others. Additionally, we need to evaluate the effect of the improved or underperformed components on the critical path before and after applying the new optimization.

Our initial critical-path analysis pinpoints three major sources of bottleneck for the baseline TFlex composable processor. These bottlenecks include (1) cross-core intra-block instruction communication, (2) block window fills following misprediction, (3) speculation overhead caused by branch misprediction. The next chapters will discuss the microarchitectural and compiler methods developed for reducing system bottlenecks. For each optimization, we repeat bottleneck analysis to detect the effect of that optimization on the critical path and other bottlenecks. This methodology helps us better desgin and select the right optimization mechanism in each step of the optimization process.

For future distributed systems, with various components each performing a particular function, performing a systematic analysis of the system bottlenecks is very important and can significantly improve the design and optimization phase of the system. Additionally, performing an automated analysis at different levels similar to the one proposed here can highly facilitate the process of bottleneck detection and removal.

Chapter 5

Strategies for Mapping Dataflow Blocks to Distributed Hardware

The first step of our bottleneck analysis indicates that cross-core communication among instructions in each block is one of the most dominant bottlenecks of the baseline TFlex system. This issue is mostly caused by the way original EDGE architectures map dataflow instructions in different blocks across distributed cores. To maximize parallelism in each block of the code, in both TRIPS and TFlex, instructions in each block are distributed across different participating cores. The location core of each instruction in a given block is determined by the compiler considering the estimated critical path of instructions within that block. Although maximizing parallelism may be important under certain circumstances, its effectiveness depends on the type and intensity of communication among the distributed instructions. In general, balancing concurrency and communication is a fundamental challenge when mapping instructions in a single-thread program to a distributed substrate of homogeneous cores. As the granularity of parallel computation increases, the frequency and cost of communication changes. This chapter re-examines the original strategy used by TRIPS and TFlex and the assumptions behind it by comparing that strategy with other possible strategies for mapping instructions into homogenous cores.

5.1 Balancing Concurrency and Communication

This chapter investigates the tradeoff between communication and concurrency for the case where the parallel unit of computation is a fixed-size block of instructions in a distributed large instruction window. Instruction-level parallelism can be exploited by mapping each block of instructions to one or more cores. Block-level parallelism can be exploited by mapping multiple blocks of instructions to the substrate at the same time.

We first consider a spectrum of *fixed* policies in which each block is mapped to a fixed number of cores. At one extreme, a *flat* policy spreads the instructions within a block across all participating cores. This flat strategy achieves high performance with single-issue cores, at the cost of frequent operand communication. At the other extreme, a *deep* strategy maps all of the instructions in a block to only one core. This strategy performs well for dual-issue cores, which are able to exploit intra-block parallelism locally while reducing operand traffic significantly.

We also explore an *adaptive* strategy, in which a compiler specified concurrency value is used to adjust the number of cores to the block. Results show that adaptive outperforms fixed strategies on both single and dual-issue cores. When running on single-issue cores, the adaptive strategy achieves higher performance than the flat strategy with operand traffic comparable to that of the deep mapping strategy. However, due to complexity and the

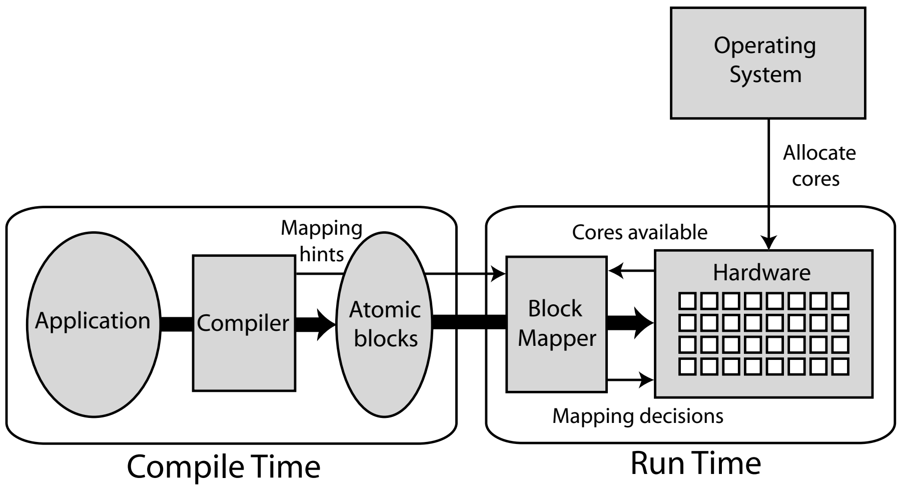


Figure 5.1: Runtime and compile time system components.

amount of implementation storage needed and for the purposes of brevity, the adaptive strategy is only explained in this chapter and the presented results only include deep and flat mapping strategies. For more details, refer to [65, 67].

5.2 EDGE Hardware Software Components

Figure 5.1 shows the components in a TFlex system. The compiler [78] breaks the program into single-entry, predicated blocks of instructions, similar to hyperblocks [52]. The EDGE ISA imposes several restrictions on blocks to simplify the hardware. We chose an implementation with a maximum block size of 128 instructions, and thus 7-bit target dataflow instruction encoding. Each block can contain up to 32 register reads, 32 register writes, and 32 load/store instructions. The compiler currently achieves about 64 dynamic instructions per block.

During compilation, the compiler's instruction scheduler generates blocks containing dataflow instructions in target form. Each instruction directly specifies its consumers using 7-bit instruction identifiers (IDs) assigned by the instruction scheduler. To generate these IDs, the scheduler takes as input the hardware topology, which includes the number of reservation stations, the maximum number of participating cores, network latencies, and a mapping of IDs to cores. For a given configuration, the scheduler seeks an assignment of IDs that minimizes the latency of the critical path through each block by minimizing communication costs along that path and exploiting available concurrency [20]. The compiler implicitly encodes the ID for each instruction in the binary via its location within the block. At runtime, the hardware routes results based on the target ID. The hardware block mapper uses IDs to map instructions to the distributed substrate. The instruction scheduler and block mapper agree upon a mapping contract and thus the scheduler can convey static information about concurrency, locality, and criticality via IDs. The next section presents more details on ID encoding.

The runtime system allocates N cores to an application based on resource availability. The hardware fetches and executes up to N blocks in parallel on the N participating cores, where N is a power of two. One executing block is always non-speculative and the others are speculative. The mapping strategy determines how many instructions from the same block a core executes. For example, a core can execute up to 128 instructions from the same block, or $128/N$ instructions from N different blocks. Inter-block communication occurs via registers, cache, and memory based on hash functions. Intra-block communication between instructions depends on the dataflow graph, the number of participating cores, and the mapping of blocks to participating cores.

5.3 Block Mapping Strategies

For a given block, the block mapper may choose to distribute the block across all participating cores, a subset of these cores, or a single core. Each strategy represents a different tradeoff between parallelism and communication overhead. We explore *fixed* and *adaptive* strategies. The fixed mapping strategies choose the same number of cores for all blocks in a program. At one extreme, the fixed *flat* strategy partitions the block across all participating cores, exploiting intra-block concurrency. At the other extreme, the fixed *deep* strategy puts the entire block on a single core, minimizing intra-block communication. The adaptive strategy seeks a better tradeoff by choosing the number of cores based on block characteristics. For each of these block mapping strategies, the block mapper interprets IDs assigned to each instruction by the compiler. We next describe this software/hardware contract in more detail, and then discuss each mapping strategy.

5.3.1 Compiler/Hardware Contract

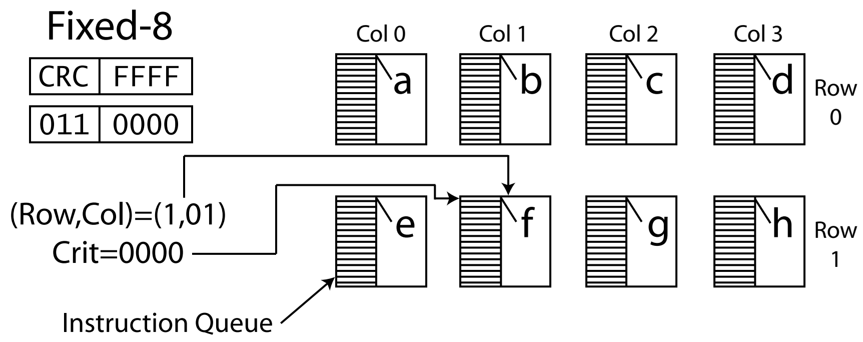
We use IDs to express criticality and locality. The block mapper reinterprets these IDs to allow programs to run on a different number of cores without being recompiled. Because there are at most 128 instructions in a block, the compiler assigns each instruction a 7-bit ID that determines *where* the instruction will execute, i.e., on which core. At runtime, instructions execute *when* their operands arrive. If two instructions on the same core are both ready at the same time, the more critical instruction should execute first. The block mapper uses the IDs to determine the order in which instructions appear in the reservation stations on each core, thus, the ID can be used to express criticality information as well as locality information.

The instruction IDs should preserve locality information if the block is mapped to a smaller number of cores. We use an abstract mapping between IDs and cores, but for ease of understanding, consider a simple mapping where IDs directly encode instruction locations. Imagine 32 cores laid out in a 4 by 8 grid, and the compiler and hardware could agree that IDs 0-3 map to core (0,0), 4-7 to (0,1), and so on. At runtime, if there are only four participating cores laid out in a 2 by 2 grid, the block mapper must interpret the bits differently, for instance by mapping IDs 0-31 to (0,0). The problem with this simple mapping is that instructions that were one hop away, those mapped to (2,4) and (3,4) in the 4 by 8 grid, are now assigned to (0,1) and (1,0), which are two hops away in the 2 by 2 grid. Ideally, the IDs should be assigned and interpreted such that instructions mapped to the same or nearby cores when compiled for N cores remain on the same or nearby cores when mapped to a smaller number of cores. We use the following abstract encoding to achieve this versatility.

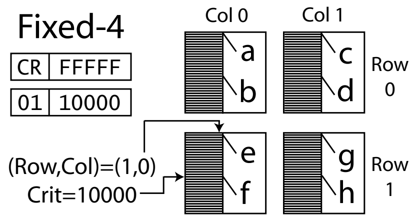
Figures 5.2(a-c) show the software/hardware contract for ID bits when running on eight, four, and two cores, respectively. With eight cores, each core will execute 16 of the 128 instructions and the first three bits determine the core. The scheduler encodes locality information in these top three bits: R (row) and C (column) in the figures. The four remaining frame (F) bits express criticality information, where lower is more critical and appears earlier in the reservation stations. The core chooses to execute the lower numbered instructions when two instructions are ready to issue in the same cycle. Similarly, mapping to four and two cores, the microarchitecture uses two and one locality bits, and five and six criticality bits, respectively.

By interleaving the R and C bits in the IDs, the compiler helps the hardware preserve locality information when mapping blocks to different numbers of cores. For example, in Figure 5.2(a), the scheduler maps dependent instructions a and b to two adjacent cores, and independent instructions a and h to two distant cores. At runtime, when mapped to four and two cores, as shown in Figures 5.2(b) and 5.2(c), the relative locality among these instructions is preserved. This format for IDs, however, does not preserve the criticality of instructions because as instructions are mapped to fewer cores, locality bits are converted to criticality bits. For example, all eight instructions in Figure 5.2(a) have high criticality and are thus placed in the highest position

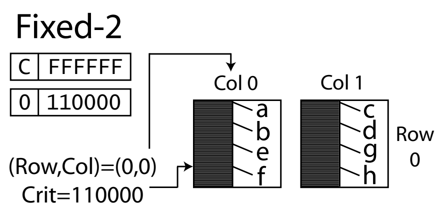
Inst	ID
a	0000000
b	0010000
c	1000000
d	1010000
e	0100000
f	0110000
g	1100000
h	1110000



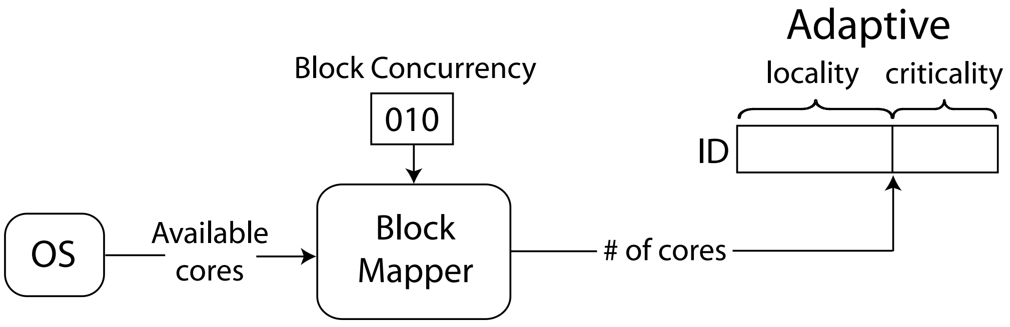
(a) Running on 8 cores



(b) Running on 4 cores



(c) Running on 2 cores



(d) Adaptive mapping

Figure 5.2: Information encoded in the instruction IDs for fixed and adaptive mapping strategies.

in their cores' reservation stations. When mapped to four and two cores, as shown in Figures 5.2(b) and 5.2(c), however, the relative positions of these instructions in their reservation stations change dramatically. Fortunately, instructions are allowed to issue out of order, so the criticality bits only become a factor when multiple instructions are ready to execute at the same time.

5.3.2 Fixed Mapping Strategies

Each fixed strategy represents a different tradeoff between communication overhead and ability to exploit concurrency. We discuss two extreme fixed strategies in this subsection.

5.3.2.1 Flat Mapping

With a flat mapping strategy, the block mapper distributes the instructions in each block across all participating cores. This approach exploits as much intra-block concurrency as possible, but incurs high intra-block communication overheads.

The IDs convey both locality and criticality information with the flat mapping strategy [67]. For example, in a 2×4 configuration containing eight total cores, each of the eight cores executes 16 of the 128 instructions as shown in Figure 5.2(a). The flat mapper uses four bits to indicate the location of the instruction, two bits for the row, and two bits for the column. The remaining three bits express criticality information – the relative issue priority that breaks ties in the reservation stations on each core (see Figure 5.2). Instructions that are close to each other when compiled to 16 cores are close, or on the same core, when executed in a flat mapping on a smaller number of cores. The TRIPS prototype employed what was essentially a flat mapping strategy across 16 execution tiles [75].

When using the flat mapping strategy, the compiler scheduler predetermines the core locations and criticality values (their order in reservation stations) of instructions in each block [20, 21]. The scheduler minimizes the latency of the critical path through the block by minimizing communication costs along that path and exploiting available concurrency [20]. The spatial path scheduling (SPS) algorithm [20] employed by TRIPS and TFlex greedily chooses to place the most critical instruction at each step in its best location (core). The best location and the most critical instruction are both determined by a single number, the *placement cost*. The scheduler uses the following max of mins approach: for each instruction, record the location at which the placement cost is minimum. Among all of the instructions under consideration, the one with the largest minimum placement cost is the most critical, so place that instruction next, at its minimum cost location.

The compiler implicitly encodes this information in the binary via its location within the block. At runtime, the hardware block mapper uses this information to map instructions into the participating cores. Several recent studies have proposed methods for achieving the best compiler strategies for instruction placements when using flat mapping. There are several issues associated with finding such a strategy statically:

- An optimum schedule, not only should consider the proximity of dependent instructions, but also should consider the proximity between critical instructions and their dependent registers because registers are distributed across executing tiles (cores). However, at compile time the register allocation [66] phase occurs before the placement which is the last compilation phase. Using some simple heuristics, the scheduler is able to relatively reduce the register read delay [66].

- Achieving optimum location and criticality for every instruction in each block not only requires the information about the local critical path within each block but also needs a more general knowledge about the global critical path for the entire period of the program execution [21]. To evaluate the limits of static scheduling using SPS, we employ Neuro-Evolution of Augmenting Topologies (NEAT) [83], which is a publicly available reinforcement learning package with an active user base that can be used to tune compiler heuristics with very little modification. NEAT successfully tuned the *placement cost* heuristic in the spatial path scheduling algorithm for individual benchmarks [21, 88, 89]. NEAT achieved performance significantly better than both hand-tuned heuristics and placements produced via simulated annealing when using specialized heuristics. Although NEAT produces good placements when specialized for individual benchmarks, finding good general solutions is very difficult. As a result, except for the highly regular applications such as matrix multiply [24] achieving the best schedule statically using a fixed heuristic seems not to be feasible.

5.3.2.2 Deep Mapping

With a deep mapping strategy, the block mapper assigns all instructions within a block to a single core. This strategy eliminates cross-core communication between instructions, but provides only as much intra-block parallelism as the issue width of the cores. Although deep mapping eliminates communication between instructions, it may increase communication between blocks because cache banks and registers are distributed across the cores.

With the deep mapping strategy, the instruction identifiers assigned by the scheduler are no longer used for locality at all – the entire instruction identifier is devoted to determining the criticality of the instruction, i.e., the instruction's priority within the core's reservation stations.

For the DFG in Figure 5.3(a), Figures 5.3(b) and 5.3(c) provide a simple example of the flat and deep mapping strategies for two blocks, $B0$ and $B1$, on a 4-core processor. Symbols a through h represent the instructions in these blocks. Registers $R0$, $R1$, and $R3$ are located in cores 0, 1, and 3, respectively. Block $B0$ reads registers $R0$ and $R1$, and writes register $R3$. Block $B1$ reads register $R3$, which is produced by $B0$, and writes register $R0$. Block $B1$ also loads a value from cache bank $D3$ located on core 3.

The value communicated between blocks $B0$ and $B1$ via register $R3$ is an example of communication between blocks, while the value produced by instruction a and consumed by instruction b is an example of communication within a block. With flat mapping, as shown in Figure 5.3(b), the instruction scheduler tries to place instructions that access registers on the same core as the corresponding register. With deep mapping, as shown in Figure 5.3(c), however, the blocks are assigned to cores dynamically in a round-robin fashion, so most register accesses go to remote cores.

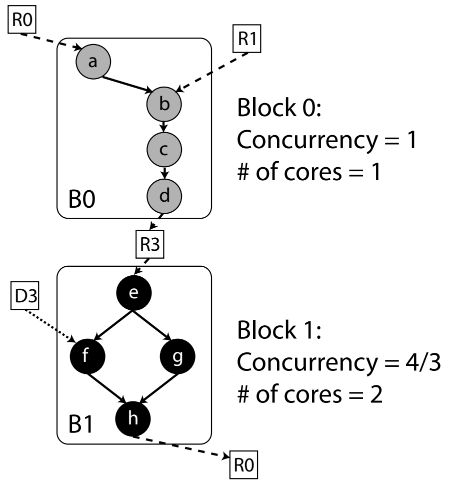
5.3.3 Adaptive Mapping

Flat and deep mapping are both limited because the block mapper selects the same number of cores, C, for all blocks in an application. The flat mapping strategy uses $C = N$, where N is the number of participating cores. The deep mapping uses $C = 1$. As a result, flat mapping may under-utilize cores or experience excessive communication overheads when blocks have low concurrency. On the other hand, the deep mapping fails to exploit all of the available concurrency for highly concurrent blocks.

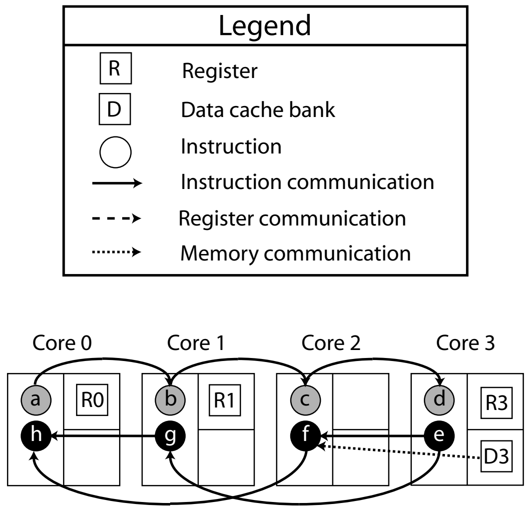
The adaptive mapping strategy balances these tradeoffs by selecting the number of cores based on the block's available concurrency and then using the IDs to map to the selected cores. The compiler evaluates the available concurrency and encodes the concurrency value in the block header as follows:

$$Concurrency = \frac{BlockInstructionCount}{CriticalPathLength}$$

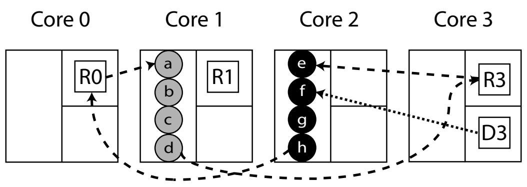
where *BlockInstructionCount* is the total number of instructions in the block and *CriticalPathLength* is the length of the critical path through the block



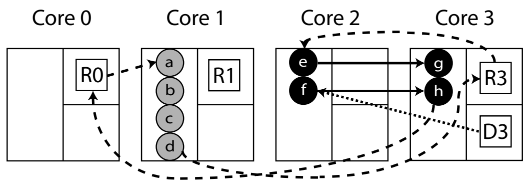
(a) DFG



(b) Flat block mapping



(c) Deep block mapping



(d) Adaptive block mapping

Figure 5.3: A sample DFG consisting of two blocks mapped using the flat, deep and adaptive mapping strategies. Solid and dotted lines represent intra and inter-block communication, respectively.

in cycles. This metric estimates the maximum achievable IPC for the block. At runtime, the block mapper dynamically selects a set of cores for the block based on the concurrency value provided by the compiler as follows:

$$C = 2^{\lceil \log_2 \lceil \frac{Concurrency}{IssueWidth} \rceil \rceil}$$

where *IssueWidth* is the issue width of each core, assuming homogeneous cores. The block mapper uses this number of cores, always a power of two, if they are available.

Using the adaptive strategy, a round-robin algorithm chooses the cores for the next block, similar to deep mapping, but it also accounts for requests with varying numbers of cores. If there is not enough room in the instruction window for the next block, then instruction fetch stalls until there is sufficient space available. More sophisticated algorithms are possible, but may make the hardware implementation impractical. Round-robin strategies can be implemented in a distributed fashion without any centralized components.

Figure 5.3(a) shows the concurrency and core count for blocks $B0$ and $B1$, and Figure 5.3(d) illustrates the adaptive block mapping for these blocks on a 4-core processor. For simplicity, this example assumes that the static execution time for all instructions is one cycle, and that all cores have an issue width of one. $B0$ consists of a chain of dependent instructions, and all of its instructions are on its critical path. As a result, its concurrency is equal to 1.0, and the block mapper assigns one core to this block. On the other hand, the length of the critical path of block $B1$ is three cycles, but this block has four instructions, which results in a concurrency value of 4/3. For this block, the number of cores chosen by the block mapper is equal to $2^{\lceil \log_2 \lceil \frac{4/3}{1} \rceil \rceil} = 2^{\lceil \log_2 2 \rceil} = 2$. If the cores were dual-issue, the concurrency values for $B0$ and $B1$ would remain the same, but the block mapper would assign one core to each of the blocks in this example.

As shown in Figure 5.2(d), the adaptive strategy uses the concurrency information for each block to select an appropriate number of cores for that block. At runtime, this number determines how many bits in the instruction identifier specify locality and how many bits specify criticality.

5.3.4 Reducing Communication Between Blocks

One disadvantage of the deep and adaptive block mapping strategies is that they may increase communication between blocks. One way to deal with this problem is to use a different algorithm to select the next core in the block mapper. We propose two possible algorithms.

Inside-Out. The Inside-Out algorithm prioritizes the cores close to the center when selecting the next set of cores at runtime. Because the cores close to the center have a smaller average distance to other cores, they should require a smaller average hop count to access registers and memory locations.

Preferred-Location. The compiler encodes a list of preferred cores in the block header. During core selection, the Preferred-Location block mapper selects the available cores highest in this list. To prioritize the cores, the compiler computes the static hop count required to access registers. For example, in Figure 5.3(d), block *B0* prefers core 1 to core 0 because core 1 will require two cycles to read *R0* and *R1*, and write *R3*, whereas core 0 will require three cycles. If cores 0 and 1 are both available for *B0*, the block mapper will choose core 1. A drawback of this algorithm is that the compiler must know the number of cores assigned to the program, making it less general than Inside-Out selection.

5.3.5 Hardware Complexity and Cost

The dynamic block mapper for deep and adaptive strategies can be implemented in a fully distributed way among cores, thus, there is no central unit for making block mapping decisions. Distributing the block mapper among cores minimizes its effect on the latency of the critical path. Here, we briefly discuss various components in this distributed block mapper.

Next core selection mechanism. The core selection mechanisms can be implemented in a fully distributed fashion. For the deep mapping strategy, the selected core for the current block sends a message to the next core in round-robin order to execute the next block. This mechanism requires no extra state in the cores. The adaptive block mapping strategy, however, requires each core to keep track of the allocation status of other cores in a table consisting of $N * log_2 N$ flip flops, where N is the total number of cores. In addition, each core requires a priority encoder to choose the next set of cores using the table.

The table and encoder incur a relatively small area overhead for each core.

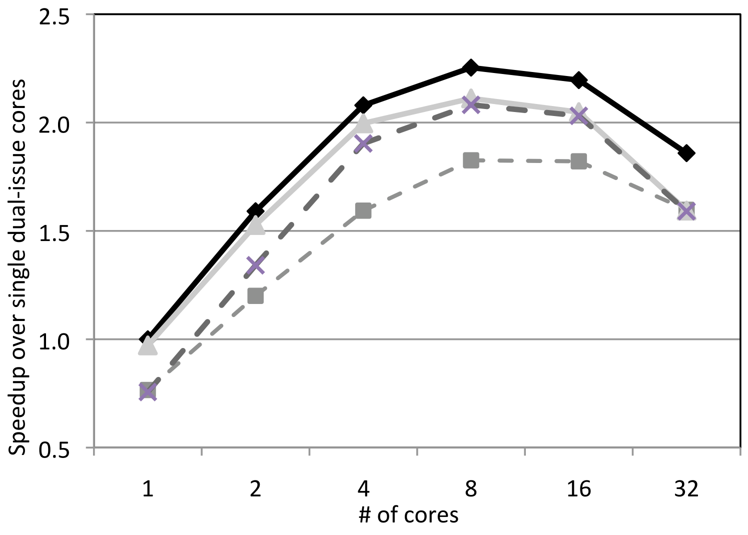
Decoding IDs. The block mapper specifies how each core interprets IDs. For example in the deep strategy, all seven ID bits determine the position of each instruction in the core's reservation stations. In the flat strategy, the mapper uses $7 - log_2N$ bits as criticality bits. In the adaptive strategy, C cores use $7 - log_2C$ bits for criticality.

5.4 Results

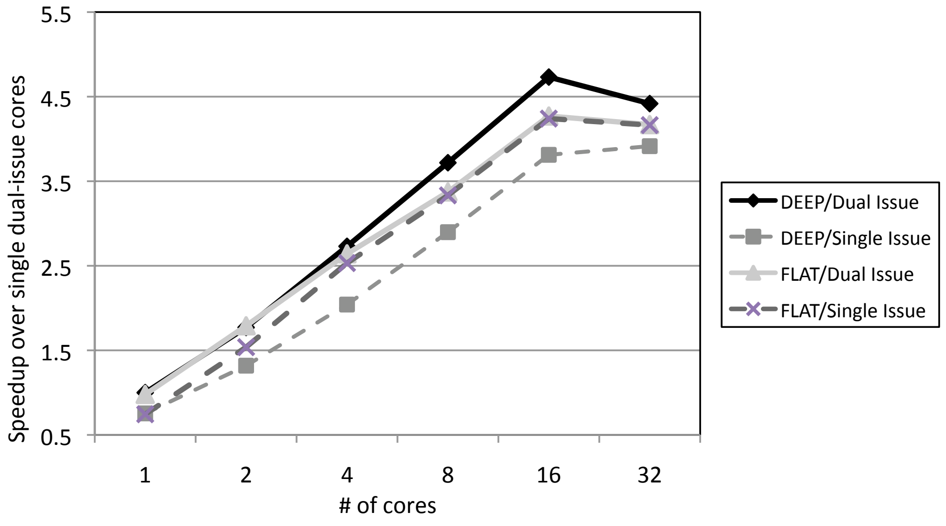
Figure 5.4 shows performance using the flat and deep mapping strategies for the SPEC benchmarks normalized to the performance of each benchmark on a single dual-issue core. These experiments vary the number of cores allocated to the application from 1 to 32 cores, and the issue width of the cores from one to two. The baseline cores, however, are always out-of-order, dual-issue TFlex cores.

With single issue cores, the flat strategy outperforms the deep strategy. However, when using dual-issue cores, the deep strategy is the best performing strategy, which indicates that dual-issue cores are enough to exploit parallelism available in each block. For the flat strategy, on the other hand, using dual-issue cores does not change the performance. That again indicates the limit of the parallelism available in each block of the distributed instruction window.

Figure 5.5 shows the percentage of executed blocks with different maximum concurrency for the SPEC benchmarks. These concurrency values are



(a) SPEC INT



(b) SPEC FP

Figure 5.4: Average speedup over a single core for the SPEC benchmarks with varying numbers of cores and varying core issue widths.

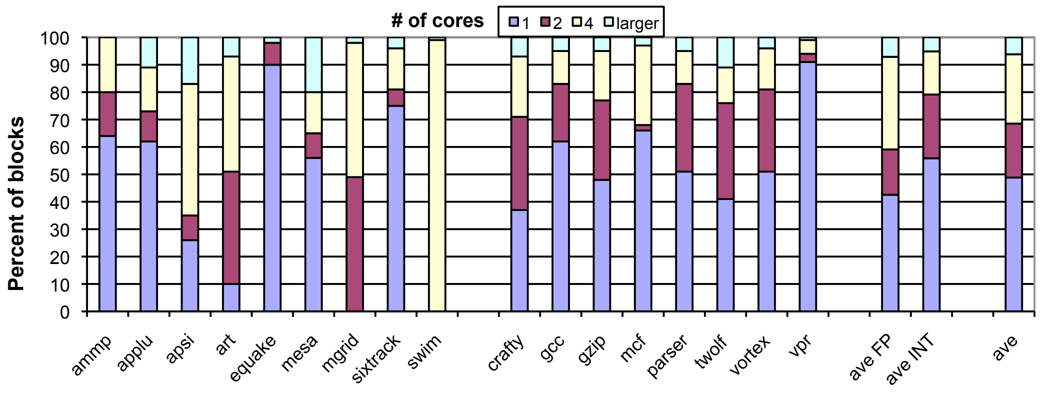


Figure 5.5: Percent of blocks with different maximum concurrency values for SPEC benchmarks.

computed by the compiler as follows. The compiler evaluates the available concurrency and encodes the concurrency value in the block header as follows:

$$Concurrency = \frac{BlockInstructionCount}{CriticalPathLength}$$

where *BlockInstructionCount* is the total number of instructions in the block and *CriticalPathLength* is the length of the critical path through the block in cycles. This metric estimates the maximum achievable IPC for the block. 80% of the blocks in the SPEC INT benchmarks can potentially achieve their maximum IPC when each is running on dual-issue cores.

Figure 5.6 illustrates the average communication overhead for each block mapping strategy for the SPEC benchmarks running on composed 16 dual-issue cores. These results are normalized to the total hop counts across all executed instructions when using the flat mapping strategy. When using

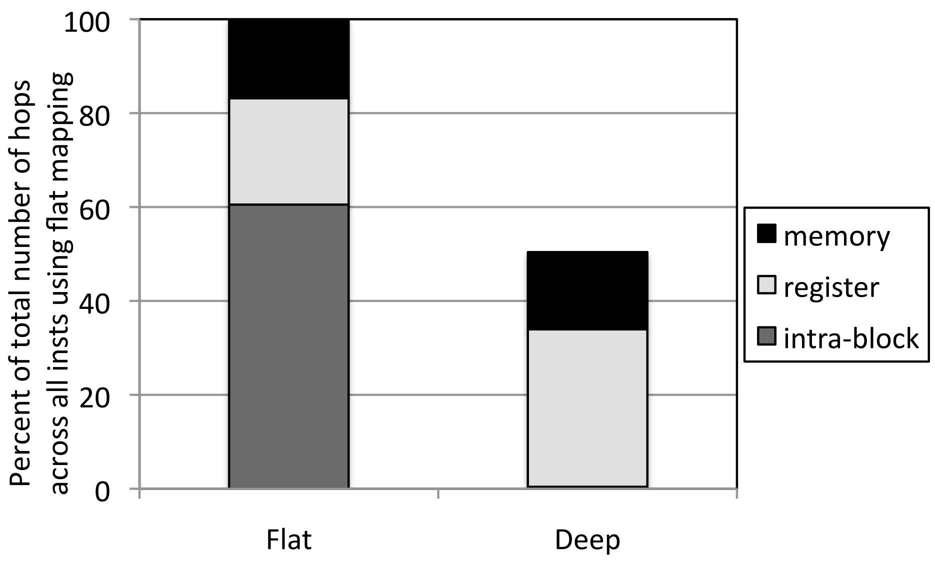


Figure 5.6: Communication overhead in terms of hop count for the SPEC benchmarks running on 16 composed dual-issue cores.

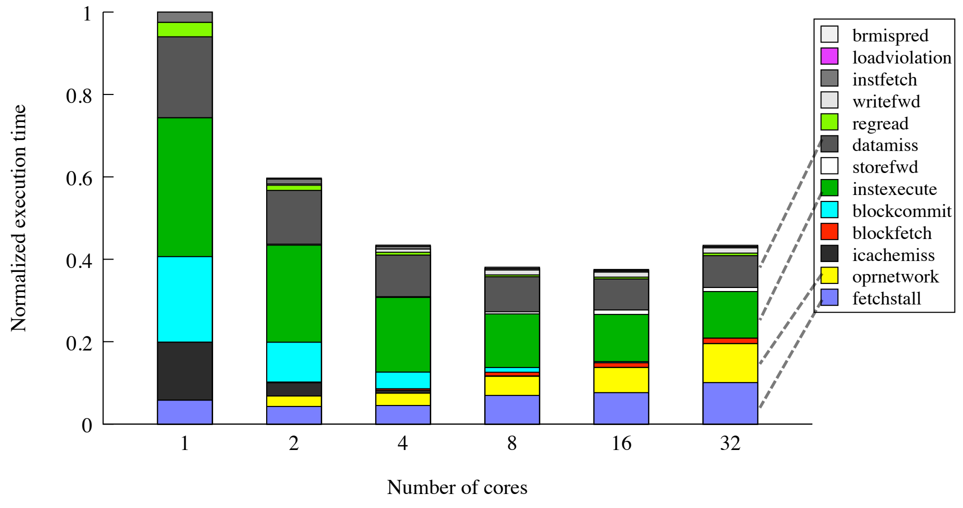
the flat mapping strategy, 60% of communication consists of dataflow operand transfer for intra-block communication. With deep mapping strategy, there is no cross-core intra-block communication and the total traffic is 50% of that of the flat strategy. Memory accesses cause almost the same amount of traffic for both mapping strategies, but the overhead of register accesses is reduced for the flat mapping strategy. The static instruction scheduling algorithm considers the location of registers on an abstract substrate when calculating the placement cost for each instruction for flat mapping, thus minimizing register latency. With the deep strategy, this traffic is almost doubled, forming almost 67% of all traffic. The critical path analysis shows that using the deep mapping strategy reduces communication significantly. Therefore, another advantage of using the deep strategy over flat strategy is its reduced inter-core communication overhead.

Most SPEC integer benchmarks reach their maximum performance when running on eight cores and observe a slowdown when running on 16 cores. Running on 32 cores, however, causes a significant reduction in performance. This results indicate possible performance bottlenecks when scaling the number of core to more than 8 cores for integer benchmarks. In the next subsection, this study uses the critical path analysis to investigate the remaining bottlenecks. For brevity's sake, we only present the criticality analysis results for the deep block mapping strategy which is the best performing block mapping strategy.

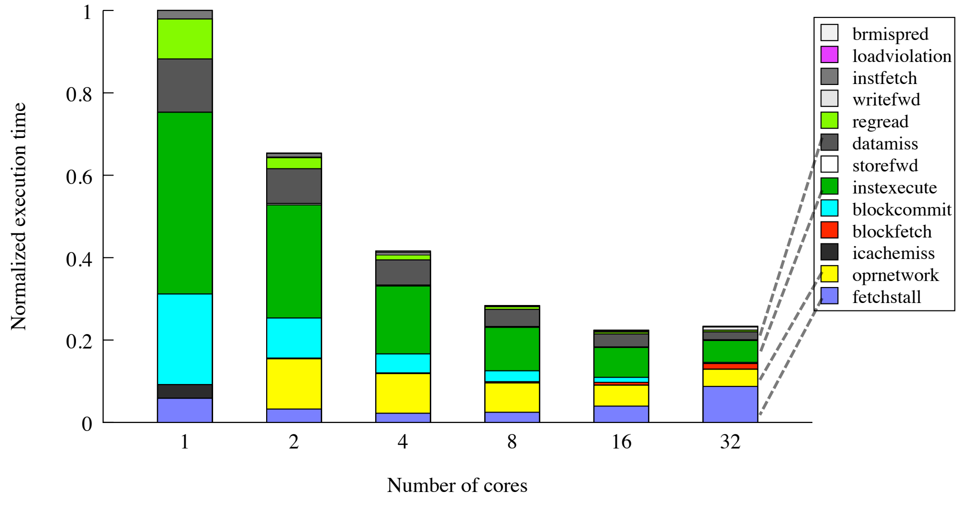
5.5 Detecting Next Dominant Bottlenecks

After reducing intra-block communication using the deep block mapping, we apply our bottleneck analysis again to detect the next bottlenecks in the system for the next iterative bottleneck reduction step. This phase of analysis is shown as the step 2 in Figure 4.5. This analysis also helps us accurately evaluate the deep block mapping strategy and its effects on different components in the system.

Figures 5.7 and 5.8 report the system-level and speculation-aware system-level breakdowns of the critical path for INT and FP benchmarks when using the deep block mapping. The dotted lines in the figure highlight the major components in the critical path. Comparing speculation-unaware results



(a) SPEC INT



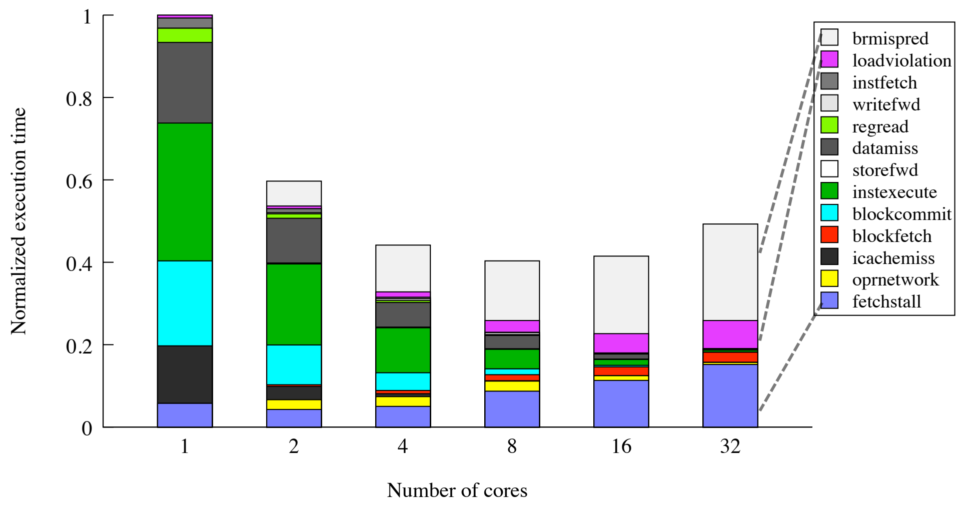
(b) SPEC FP

Figure 5.7: Critical path breakdown for different micro-architectural components when applying *deep* block mapping.

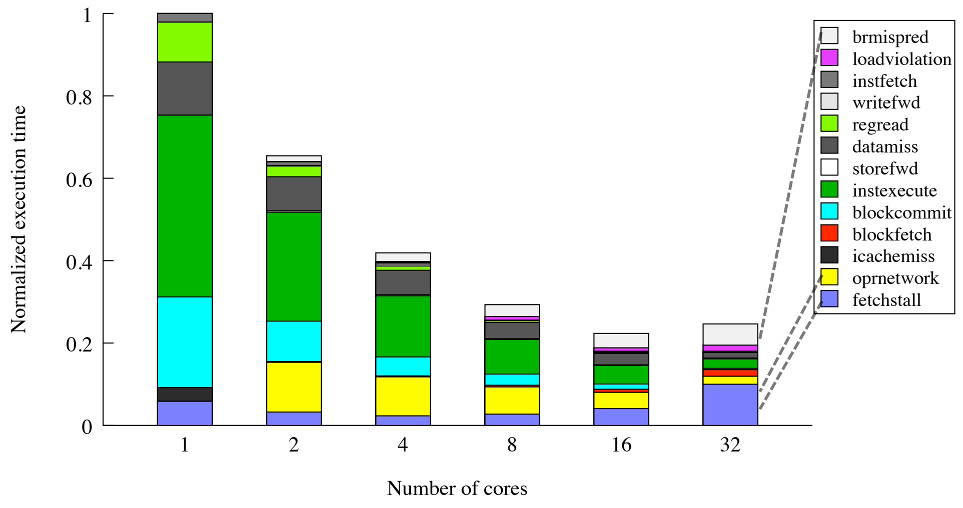
shown in Figures 5.7 and 4.2 shows that the deep mapping model reduces the effect of the cross-core communication bottleneck to some extent. However, the fetch bottleneck becomes more dominant when using this optimization, which limits the speedup achieved. Comparing Figure 5.8 and Figure 4.3 shows that the deep mapping model reduces the overhead caused by misspeculation in high core-count configurations. This indicates that reducing intra-block communication speedups execution of instructions leading to the detection of a misprediction. However, the events following misspeculation events become more critical. That is why fetch stalls are more pronounced on the critical path when using the deep mapping optimization.

Figure 5.9 illustrates the component-level results for the on-chip network and fetch components when applying the deep mapping strategy for INT benchmarks. For INT benchmarks, when using eight or more cores, *data misses, instruction execution, on-chip inter-core network* and *fetch stalls* are the major contributors of the critical path. Among these factors, the contributions of the *on-chip network* and *fetch stalls* (the lowest two segments in each configuration shown) still increase as more cores are merged. The following two components can be considered as the system's main bottlenecks:

On-chip Network: When applying deep mapping, the on-chip network is only used for inter-block communication between instructions running on different merged cores. Figure 5.9(a) shows that in absence of intra-block cross-core communication, register communication becomes more dominate and occupies most of the critical cycles on the on-chip network. This result

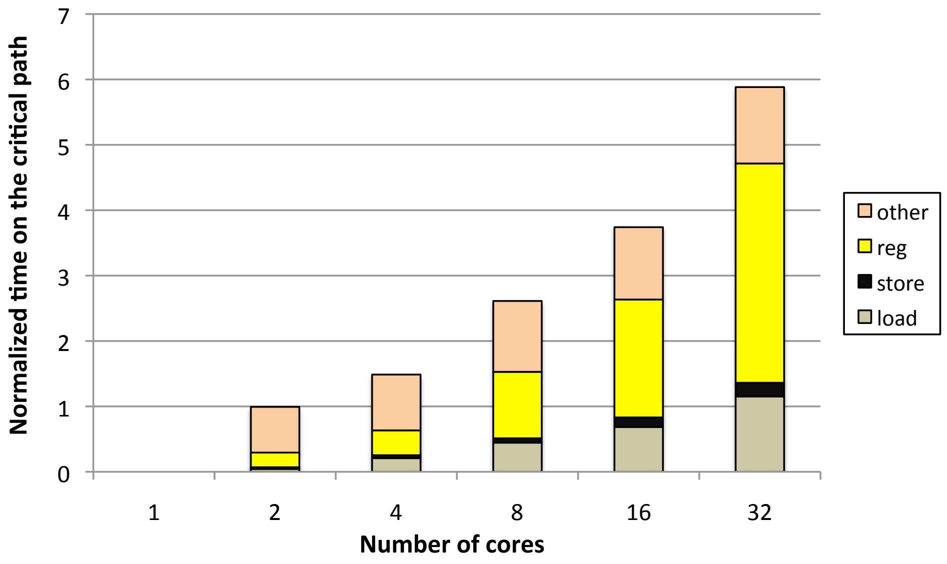


(a) SPEC INT

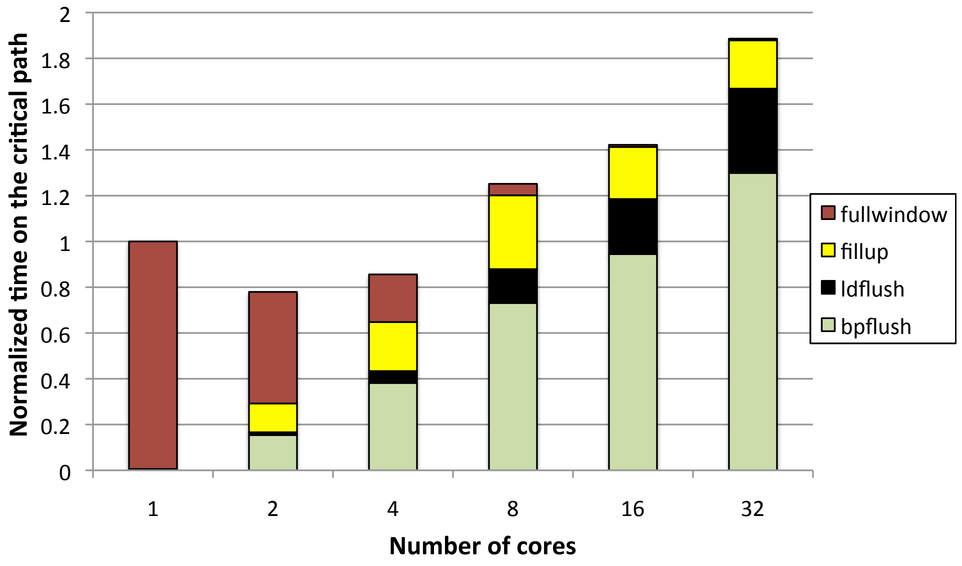


(b) SPEC FP

Figure 5.8: Speculation-aware critical path breakdown for different micro-architectural components when applying *deep* block mapping.



(a) Onchip Network



(b) Fetch

Figure 5.9: Breakdown of critical cycles for on-chip network and fetch bottlenecks for *SPEC INT* benchmarks using *deep* block mapping.

shows that most critical network traffic is caused by register communication between distributed instructions. A producer instruction sends a new value of an architectural register to the home core of that register. Resolving dependences, the register forwarding unit of that core then forwards the value to its consumer core(s). When the core count increases, the network distance increases and so does the average inter-block communication delay.

Fetch: Figure 5.9(b) does not show major changes in the breakdown of fetch critical cycles when using the deep block mapping strategy. Comparing this figure to Figure 4.4(b), however, shows that the number of critical fetch cycles increases for the configurations with high numbers of cores as a result of this optimization. When a large number of cores are merged, the system constructs a large window of speculative instructions. Consequently fetch stalls caused by misspeculation flushes are likely to end up on the critical path and become a performance bottleneck. This phenomenon explains the increase in fetch stall segments of the critical path when merging more cores.

5.6 Summary

This chapter explores various strategies to dynamically map blocks of instructions to a distributed hardware substrate consisting of composed cores acting as a single processor. A run-time block mapper, implemented in hardware, maps instructions to cores. We explore a spectrum of *fixed* policies, in which a block mapper maps each block of instructions to the same number of cores. At one extreme, a *flat* mapping policy partitions the instructions in each block among all participating cores, emphasizing intra-block parallelism, but increasing intra-block communication. At the other extreme, a *deep* mapping policy maps all of the instructions in a block to a single core, but successively maps blocks to different cores. The deep strategy minimizes intra-block communication delays, but allows no intra-block parallelism beyond the issue width of the individual cores, and makes inter-block communication more expensive.

For single-issue cores, like the ones used for TRIPS, a flat mapping policy is the highest-performing fixed choice. Although the flat mapping policy increases the processor's complexity and communication overheads, single-issue cores need the additional intra-block concurrency that the flat mapping provides. The low additional complexity of dual-issue cores, however, harvests enough of the intra-block parallelism to change the ideal mapping to a deep mapping. The deep mapping eliminates substantial intra-block operand communication, and the dual-issue cores provide enough intra-block parallelism that a flatter mapping provides no benefit. Both of these policies are limited, however, because they are fixed: each block is mapped to the same number of cores, regardless of the variance in ILP across different blocks. Considering these observation, we propose the *deep* strategy as the efficient strategy for the T3 microarchitecture.

Our bottleneck analysis shows when using this strategy, the intra-block communication is no longer the dominant bottleneck in the system. However, in absence of this bottleneck, the inter-block register communication and block fetch become the major bottlenecks in the system. The following two chapters propose microarchitectural optimizations to address these two bottlenecks.

Chapter 6

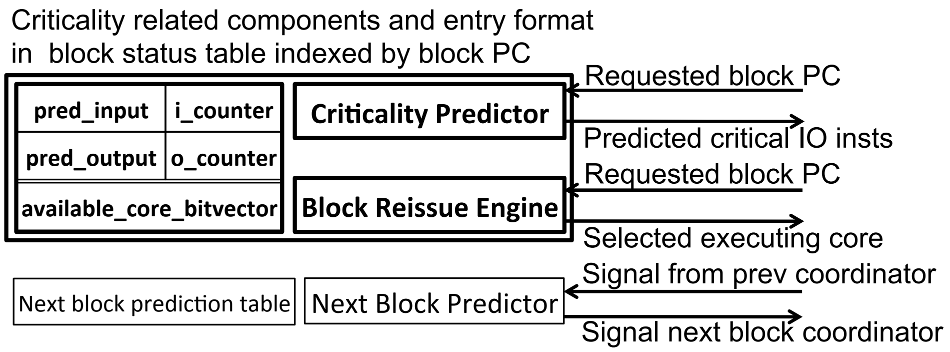
Critical Inter-Block Value Bypassing

Our bottleneck analysis shown in Figure 5.7 indicates that the most critical bottleneck in the TFlex substrate after applying deep mapping, is the coarse-grained, inter-core register communication, which occurs through shared forwarding units. To alleviate this bottlenecks, this chapter proposes and evaluates a distributed framework called *Distributed Block Criticality Analyzer* (DBCA) that exploits different types of criticality information collected at block boundaries to implement low-overhead optimizations at fine or coarse execution granularities. This general and flexible framework is implemented in a fully distributed fashion across multiple cores. Such a framework can be used for exploiting different types of criticality to optimize applications dynamically in future distributed systems. Although the proposed framework is general, for the purpose of this study, we only focus on the communication criticality and fetch criticality. The communication criticality and fetch criticality are addressed in this chapter and the next chapter, respectively.

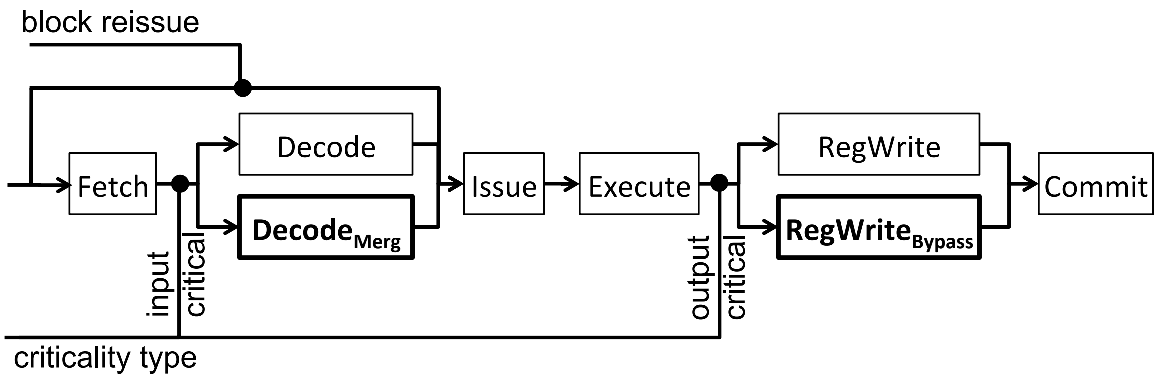
DBCA predicts critical communication instructions at block boundaries using a low-overhead criticality predictor located in a coordinator core, which is not necessarily the same as the core executing those instructions. After these critical instructions are predicted, they are selectively optimized according to their criticality types at a pipeline-stage granularity in their executing. An optimization called *selective register value bypassing* discussed in this chapter, sends values directly from each output-critical instruction in one executing core to their consumer instructions in other cores, thus bypassing shared register forwarding units.

6.1 Communication Criticality Predictor

This section explains a distributed block criticality analyzer (DBCA) used by T3 that exploits criticality information to optimize critical instructions or code blocks based on their criticality characteristics. In this dissertation, DBCA is restricted to cross-core communication and fetch criticality. However, it can be extended to include other types of criticality such as memory and execution criticality. Figure 6.1 highlights the components added to each T3 core by this analyzer. To minimize the communication overhead, DBCA piggybacks on the next block prediction distributed protocol. Each block is assigned a fixed core as its coordinator core, which is selected based on a few low-order bits of the PC of the first instruction in the block (block PC). The coordinator core contains next block prediction tables for all of the blocks assigned to it. When a new block is requested, its coordinator core is signaled by the coordinator of the previous block to allocate an idle core (a core not executing any block) to execute the new block and predict the next block and then signal the coordinator core of the predicted block.


Criticality related components and entry format
in block status table indexed by block PC
pred_input
i_counter
pred_output
o_counter
available_core_bitvector
Criticality Predictor
Block Reissue Engine
Requested block PC
Predicted critical IO insts
Requested block PC
Selected executing core
Next block prediction table
Next Block Predictor
Signal from prev coordinator
Signal next block coordinator

(a) Tables and components added for coordinating.



(b) Augmented instruction pipeline.

Figure 6.1: Components used in the distributed block criticality analyzer to reduce bottlenecks in T3.

DBCA extends this protocol by augmenting the coordinator core with a table called *the block status information* table shown in Figure 6.1(a). This table contains the criticality information of blocks assigned to this coordinator core and is maintained by two hardware components located on the coordinator core. A communication criticality predictor predicts both the communication-critical instructions and their *criticality type* and a block reissue component maintains the information required for reissuing the non-running instances of fetch-critical blocks. When a block is allocated, the corresponding coordinator core accesses these hardware components, extracts and sends the criticality information of that block to the selected executing core. The executing core uses that information to optimize the pipeline of critical instructions according to their criticality types. Communication critical instructions are treated specially in a fine-grained manner in the pipeline according to their predicted criticality type. Output-critical instructions go through a value bypassing stage which sends the produced critical register values directly from their producer cores to their consumer cores, thus bypassing the shared register forwarding units. For reissued fetch-critical blocks as will be discussed in the next chapter, all instructions skip their fetch and decode stages in a coarse-grained manner.

Note that in this distributed framework, coordinator and executing cores do not have to be physically separated and a core executing a block can simultaneously act as the coordinator core of other blocks. Table 6.1 shows coordination and execution orders in a system running 4 iterations of a loop across 8 cores. Each iteration has 2 blocks A and B (block PCs) assigned

Table 6.1: An example of mapping 4 loop iterations each with 2 blocks A and B, across 8 cores (C_1 to C_7).

Fetch order Block$_{iteration}$	A_1	B_1	A_2	B_2	A_3	B_3	A_4	B_4
Coordinator cores	C_0	C_1	C_0	C_1	C_0	C_1	C_0	C_1
Executing cores	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7

to coordinator cores C_0 and C_1, respectively. If all cores are idle at first, the coordinator cores select idle cores in a round-robin fashion for running the iterations.

6.1.1 Communication Criticality Predictor

Predicting critical communication instructions of each block can be done using a state-of-the-art criticality predictor [28] explained in Section 2.3. Although proven effective [28], using such a criticality predictor in a distributed multicore system can have high hardware complexity, communication, and storage overheads. In this subsection, we describe a low-overhead communication criticality predictor used by DBCA. For each block, *block inputs* refer to the register operands used by instructions in that block, but produced by other blocks. *Block outputs*, on the other hand, refer to the register operands produced by the instructions in that block but used by other blocks in the window. As long as all inputs of a block have not arrived from previous blocks, some instructions in that block remain uncompleted. Finally, the block cannot commit until all its outputs are sent to other blocks. Therefore, late communication edges (last-departing register outputs produced by a block before the block commits or the last-arriving register inputs received by a block) are likely to be on the critical path. To verify this, we use the critical path analysis discussed in Chapter 4 to find the breakdown of the critical communication edges. In this breakdown, the critical communication edges are divided into four categories: last-departing outputs, non-last-departing outputs, last-arriving inputs and non-last-arriving inputs. Figure 6.2 presents this breakdown for the SPEC2K benchmarks running across 16 merged cores (a few benchmarks are missing due to critical path tool complications). Note that the critical output and input edges are the edges on the critical path from a producing core to the corresponding forwarding unit on the home core of the corresponding register and from the forwarding unit to a consuming core, respectively. For INT benchmarks, 70% of all register critical inputs and outputs are late communication edges (the sum of the first and third segments from below in each bar). For FP benchmarks, late communication edges may be less critical because only 52% of critical register inputs and output are late.

Given the high criticality of the late communication edges, to reduce overheads, the predictor used by DBCA predicts late communication edges instead of critical communication edges. We only explain the algorithm for predicting the last-arriving register inputs of each block; predicting last-departing outputs is similar. The coordinator core stores late input predictions for its assigned blocks (*pred_input* in Figure 6.1(a)). For critical register inputs, the actual predicted value is the register number associated with the last-arriving

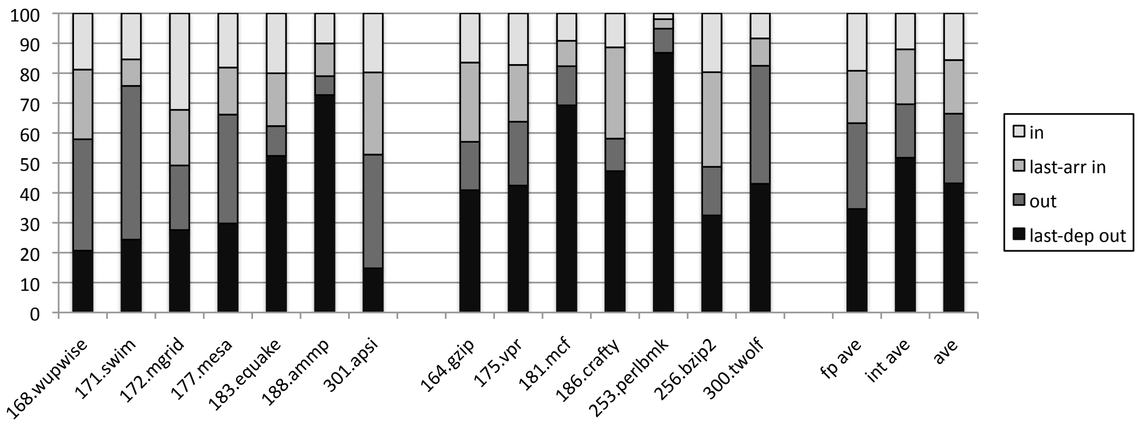


Figure 6.2: Critical-communication edges breakdown for SPEC benchmarks.

register input of the block. When a block is allocated and mapped to a core, its coordinating core predicts the last-arriving register input of that block and sends it to the core executing that block. When the block ends execution, its executing core appends the last-arrived input observed during execution to a *dealloc* message and sends it to the coordinator core along with other data needed for deallocation. The coordinator core updates the prediction entry of that block using Boyer and Moore's majority vote algorithm [12]. Each entry in the table includes the predicted last-arriving input for a block and a majority vote counter (*i_counter* in Figure 6.1(a)). When the predictor updates the entry, if the new last-arriving input is the same as the predicted one, the majority counter is incremented. Otherwise, the counter is decremented but the predicted input does not change. If the counter reaches zero, the predicted input will be updated by the current last-arriving input. To reduce the effect of the stale data in the prediction table, the predictor uses an epoch-based algorithm. This algorithm uses two prediction entries per block, each with an input number and a majority counter. During each fixed epoch, the algorithm uses one of the two entries for training and the other entry that was trained in the previous epoch, for predicting. When an epoch ends, the two entries are switched. Our accuracy evaluation of this predictor shows that when running the SPEC benchmarks across 16 cores, late register inputs and outputs of blocks can be predicted correctly 80% of the time.

6.1.2 Selective Register Value Bypassing

Before we discuss the register bypassing used by executing cores for critical instructions, we briefly discuss the the original register forwarding mechanism used by TFlex. The original mechanism uses distributed forwarding units in participating cores to resolve inter-block instruction register dependences. As each speculative block is allocated, it allocates an entry for each of its output registers in the forwarding unit of the *home core* of that register. Also, it sends its register read requests to corresponding home cores of its input registers. When a block produces a register output, the core running the block sends the output value to the home core of the corresponding register. In the home core, the register forwarding unit accesses the forwarding entry allocated to that register to look up the destination cores and instructions before sending the value. In this mechanism, the inter-block register dependences are resolved at the home core. This forwarding mechanism is *indirect* because it does not provide direct communication between source and

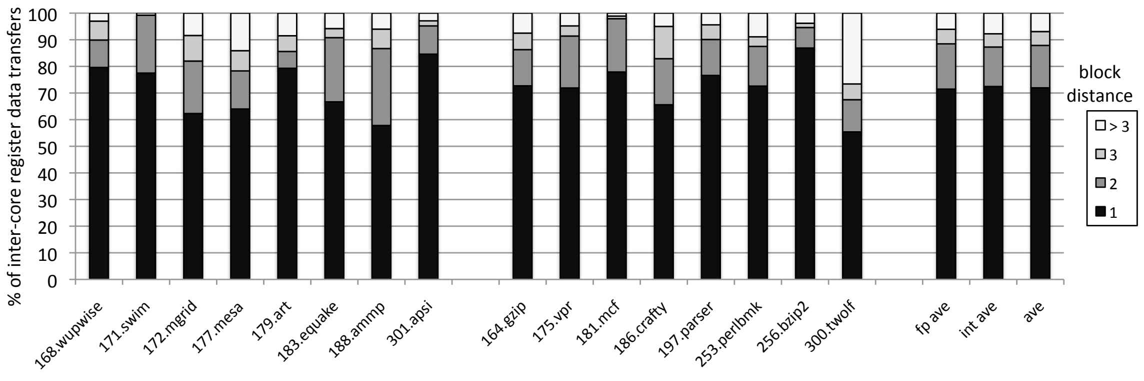


Figure 6.3: Block distances between register data producer and consumer blocks running on 16 merged cores.

destination cores. The communication delay associated with this forwarding increases as the network grows larger. An alternative communication model uses direct communication between different blocks. In this model, register dependences are resolved in the producer cores and each producer instruction in one core sends its output directly to the consumer instruction(s) in other cores. Such a mechanism requires that each core maintain a synchronizing scoreboard table [46] that tracks the status of all shared registers. To keep the tables updated and coherent, cores need to be connected though a shared broadcast bus, which can reduce scalability as high numbers of cores are used.

DBCA uses a low-overhead, direct communication mechanism called *value bypassing* for critical output instructions. However, other instructions use the original indirect forwarding mechanism. We restrict register value forwarding to only immediate successive speculative blocks. Figure 6.3 shows block forwarding distances between producer and consumer blocks for the SPEC benchmarks running on 16 cores. The block forwarding distance in this figure is the number of speculative blocks between a producer block and its consumer blocks. On average, 74% of the value forwarding happens between two subsequent speculative blocks. Considering the aforementioned simplifications, the bypass mechanism for critical registers no longer needs to track the status of all registers using synchronizing scoreboards. When the last-departing register of a block is predicted, the core executing the block sets a flag if the subsequent block reads that register. When the last-departing value is produced, the producer core sends the value directly to the core executing the next speculative block if the flag is set. The destination core forwards the value to its instructions waiting for the that register value. The value also needs to be sent to the home core of that register so that non-critical consumer cores can also receive it through the original forwarding mechanism.

6.2 Results

Figure 6.4 shows the speedups achieved using selective register value bypassing for SPEC benchmarks when using 16 merged cores. To control the number of forwarded critical values per block, we use a parameter called *criticality factor*. This factor determines the number of forwarded last-departing register values per block. For instance, using a criticality factor of one means that every block bypasses its last-departing register output to the core running the subsequent speculative block. In this graph, *bypass cfactor 1* to *3*

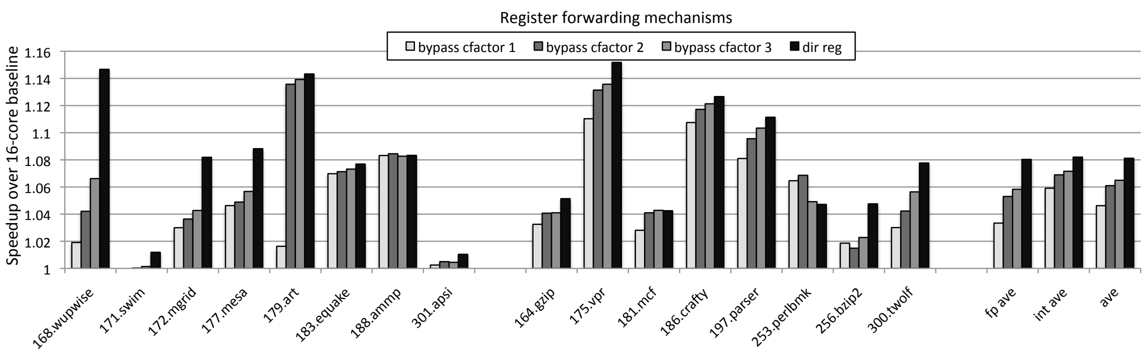


Figure 6.4: 16-core speedups achieved using value bypassing.

represent the selective bypass runs with criticality factors of 1 to 3. *Dir reg* represents a high-overhead register forwarding mechanism in which all producers forward their values directly to their consumers in all consecutive blocks. *Dir reg* can be used as an upper limit for measuring value bypassing speedups. For SPEC-INT benchmarks, the maximum speedup over the original bypassing model on average is 8%, which is achieved using *dir reg*. Only bypassing one last-departing register output of each block will achieve about 6% speedup on average. This speedup increases to more than 7% as we raise the criticality bypass factor to 3. For FP benchmarks, the maximum speedup is about 8%, while using criticality factor of 1 results in 3% speedup. Raising the criticality factor to 3 increases the speedup to 6%. As shown in Figure 6.2, the last-departing register values are not as critical for FP benchmarks as they are for INT benchmarks.

6.3 Summary

To alleviate EDGE cross-core register communication delay, this chapter proposes a flexible framework for exploiting different types of instruction criticality in a distributed dynamic multicore system. This framework augments each core with several very low-complexity distributed components and implements a distributed protocol to optimize different types of critical instructions at different levels of pipeline across cores. This framework detects critical outputs of each block using an MJRT epoch-based majority vote algorithm. The predicted critical output instructions are prioritized over other outputs and go through a fast register forward stage for communicating their results to remote cores. This mechanism bypasses the critical values directly to destination cores; thus skipping the distributed register forwarding units used by T3 for resolving registering dependences.

Chapter 7

Reducing Pipeline Flush Penalty using Block Reissue

7.1 Introduction

EDGE compilers detect useful global re-convergent points and combine basic blocks statically to create large predicated blocks. The support for predicated blocks in TFlex reduces the fetch criticality for the instructions within each block because of the bulk fetch of the block's instructions. However, as the number of in-flight blocks increases, block fetch becomes a system bottleneck as the blocks immediately following a misspeculation become fetch-critical.

The bottleneck analysis shown in Figure 5.7 indicates that the second most dominant critical bottleneck in the TFlex after applying deep mapping, is a fetch bottleneck caused by mispredictions. To alleviate this bottleneck, this chapter exploits the *Distributed Block Criticality Analyzer* (DBCA) framework proposed in the previous chapter. To reduce fetch criticality, the coordinator core maintains availability status of previously fetched blocks and reissues those blocks without fetching them again if needed. Consequently, this method saves latency and energy by short-circuiting the fetch and decode operations of all instructions in the reissued block.

7.2 Tracking Previously Executed/Flushed Blocks

The block reissue component in T3 tracks and reissues instances of blocks previously-executed in the distributed instruction window. Different from trace processors [54, 71], this *block reissue* mechanism does not rely on complex hardware for tracking and combining the blocks inflight and finding global re-convergent points. Instead, it relies on the compiler to detect re-convergent points and create large predicated blocks by combining basic blocks. All instructions in each block remain in the instruction window of the executing core until the block commits or is flushed. So, distributed instruction queues on the participating cores can be used as intermediate instruction storage. Also, by shortcutting critical fetch and decode operations as shown in Figure 6.1(b), the mechanism achieves energy savings. For example, when the first iteration of the loop in Table 6.1 commits, A_1 and B_1 available instances of blocks A and B in the instruction queue can be immediately reissued to run iteration 5 on cores C_0 and C_1.

This power saving is not limited just to reissuing control-independent blocks. This mechanism can also reissue other frequently executed blocks in a program. For example when running a large loops, the program needs to fetch the same group of blocks for each iteration of the loop while most of those blocks can be reissued as they are in the instruction queue.

7.3 Reissuing Blocks

To support block reissue, a coordinator core stores a bit vector (*available_cores_bitvector* in Figure 6.1(a)) for each block assigned to it, which represents the idle cores in which a non-running copy of the block is available. When a block is allocated/committed, its coordinator core resets/sets the bit corresponding to the executing core of that block. When a block is requested, its coordinator core searches the corresponding bit vector to find a core with a non-running instance of the block and reissue that block and reset the corresponding bit in its bit vector. If the block is not available in any of the idle cores, the executing core selects an idle core to fetch the block from i-cache and execute it. In this case, the selected core deallocates its previously-executed block and informs the coordinator core of that block to update the bit vector of the deallocated block. To increase the hit rate of the block reissue mechanism, we can use more than one instruction queue in each core. For instance, when using two instruction queues per core, each core can store up to two decoded blocks. At a given time, however, each core can only execute one of its two stored blocks and the other instruction queue is used as an instruction storage. Studying locality of reissued blocks and exploring different search and replacement policies is a part of the future work of this dissertation.

7.4 Results

For the block reissue experiments, we implement an LRU block replacement policy and a first-match search on the *available_cores_bitvector* bit

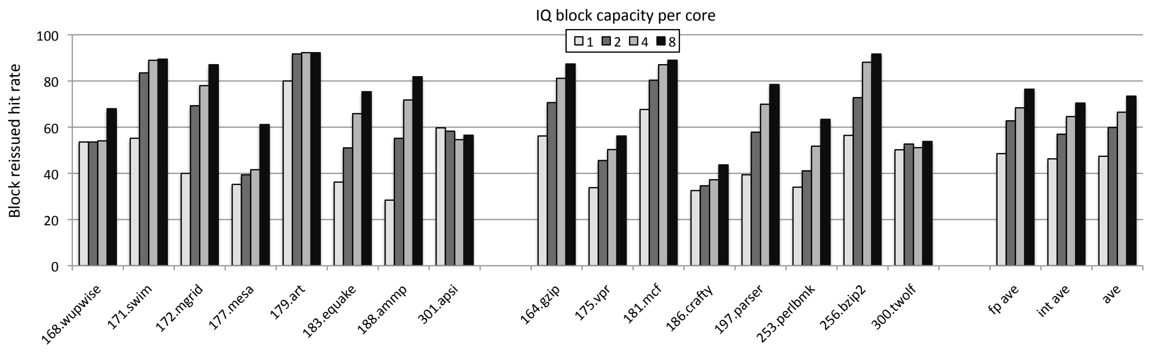


Figure 7.1: 16-core block reissue hit rate with varying instruction queue block capacity per core.

vector of each allocated block. More complex search and replacement policies can improve reissue hit rate of fetch-critical blocks. Figure 7.1 reports the block hit rates for our block reissue mechanism using 16 merged cores for SPEC benchmarks. When storing one block in the instruction queue of each core, the reissue hit rate is about 50%, which translates to decreasing the energy consumed by fetch and decode by half. Doubling the instruction queue storage (with only one block running), the reissue hit rate increases to 60%. Raising the storage size from 2x to 8x results in a hit rate of more than 75%. The hit rate is higher for FP benchmarks most probably due to their smaller code size and abundance of loops. Figure 7.2 reports the 16-core execution times normalized against a 16-core baseline which does not use block reissue. For some benchmarks, we observe a slowdown when applying block reissue. Our investigation shows that for these benchmarks, block reissue has a negative effect on the accuracy of the data dependence predictor. Ignoring those

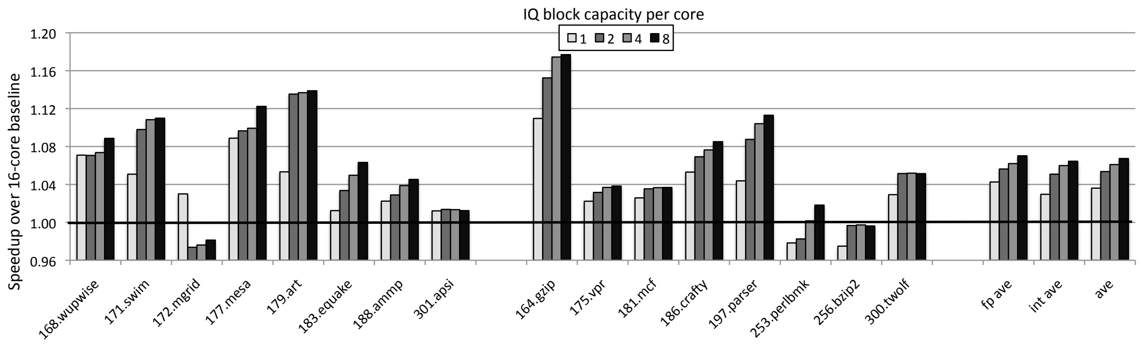


Figure 7.2: 16-core block reissue speedups with varying instruction queue block capacity per core.

Table 7.1: Percentage breakdown of all reissued blocks with instruction queue block capacity of two.

Benchmarks	After branch mispredictions	After load violations	Single block loops	others
INT	54.4%	20.0%	1.9%	23.7%
FP	26.0%	9.8%	11.9%	52.3%

benchmarks, average speedups across INT and FP benchmarks for different block storage size per core are very similar. The reissued blocks come from different sources on the INT and FP benchmarks. The hit rate is higher for FP benchmarks most probably due to their smaller code size and abundance of loops. Table 7.1 includes the breakdown of reissued blocks for the runs with instruction queue block capacity of two. For INT benchmarks, the majority of the reissued blocks are reissued after block misprediction events. For FP benchmarks, large loops and other repetitive code patterns comprises the majority of reissued blocks.

7.5 Detecting the Next Dominant Bottlenecks

The section presents the critical path analysis the system after having cross-core register communication and fetch bottlenecks reduced. Figure 7.3 compares the critical path pattern for different optimization mechanisms so far proposed in this dissertation when applied iteratively in the three discussed steps. Each bar is labeled by an optimization mechanism added in a optimization step. The right most bar in this graph shows the results for our analysis including all three used optimizations for the 16-core configuration normalized against the 1-core results. For brevity, we do not report the results for other configurations and core counts. The steps shown in the figure are our bottleneck analysis steps shown in Figure 4.5. The communication and fetch stall segments of the critical path (the two lowest segments of each stack) are significantly reduced after applying the register bypassing and block reissue mechanisms. The new critical path comprises mostly instruction execution. The other important component is data misses. This criticality pattern is nearly ideal for a system with distributed partitions of a large instruction window. According to this graph, most useful cycles of a program execution are now spent on instruction execution and data misses.

These results indicate that the last remaining bottleneck in the system is *instruction execution.* Although instruction execution is not a scalability bottleneck but it is still a large critical component when merging a large number of cores. The component-level analysis indicates the execution is now limited only by program data dependencies, including intra-block predicates,

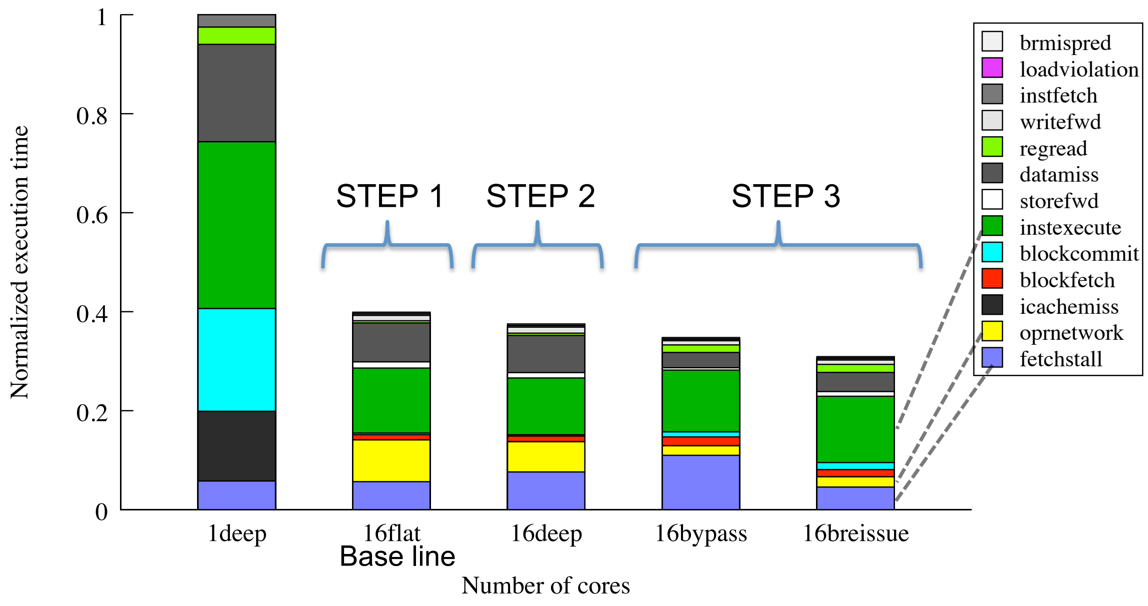


Figure 7.3: System-level breakdown when applying deep block mapping, register bypassing and block reissue.

and next block prediction accuracy. As a result, even further performance can be squeezed by extending the framework for execution criticality and branches.

When running SPEC INT benchmarks across 16 TFlex cores, a large portion of critical cycles is spent by the instructions waiting for predicate values (branches converted to predicts). Additionally a significant portion of executed instructions are dataflow fanout instructions inserted by the compiler for high fanout instructions. Finally our speculation-aware results (not shown here for brevity) indicate that the speculation bottleneck is also more dominant that before. The next two chapters address these new bottlenecks.

7.6 Summary

At the block level, the DBCA framework implements a mechanism that reissues blocks of instructions while they are still in the instruction window. The results show that this mechanism reduces the number of expensive fetch and decode operations by at least 50% translating to significant energy and delay savings. This framework can be implemented on other distributed systems and can exploit other criticality types. Moreover, coarse-grained grouping of related instructions at compile-time similar to the one used in this study can simplify the implementation and reduce its overheads.

The bottleneck analysis shows that the optimization mechanisms proposed in this chapter and the previous chapter successfully reduce register communication and fetch bottlenecks. Moreover, the analysis shows that at this point, the time spent waiting for critical predicate values and execution of move instruction pose a dominant execution bottleneck on the system. Additionally, low next block prediction accuracy also limits performance.

Chapter 8

Efficient Distributed Speculation using Iterative Path Prediction

8.1 Introduction

By exploiting predicated atomic blocks, EDGE architectures do not suffer from control bottlenecks when distributing computation. To scale, EDGE architectures rely on speculative execution of several of these blocks. Performing block-level speculation requires predicting the next block when fetching each block. However, as each EDGE block is multi-exit block, an accurate next block predictor needs to predict which of these exits from the block will be taken. Given that the intra-block control flow points are converted to predicts, the global history of branches no longer includes those branches, which degrades the accuracy of the next block predictor. Also, the branches that are converted to predicates are evaluated at the execution stage instead of being predicted similar to how they are handled in conventional architectures.

This chapter proposes a mechanism called Iterative Path Prediction (IPP) to alleviate these major overheads. IPP quickly predicts an approximate multi-bit predicate path through each block and uses this predicted path combined with the global history to predict the next-block block address. The predicted path is also used to speculatively execute the predicates within the block.

The next block prediction accuracy is also affected by the compiler policies employed to merge basic blocks and generate EDGE predicated blocks. Additionally encoding fine-grained control flow information into the ISA, the compiler can potentially improves the next block prediction accuracy. This dissertation only focuses on the microarchitectural methods for improving EDGE next block prediction and predication. A followup study can analyze the sensitivity of the propose hardware mechanisms in this Chapter to the compiler policies used to form blocks.

8.2 Prediction and Predication Overheads

The EDGE compiler uses predication to generate large blocks by converting multiple nested branches into predicates. Therefore, all control points within a block are converted into predicated values generated by dataflow test instructions. By speculatively executing several of these large predicated dataflow blocks, the EDGE microarchitectures can reduce fetch, prediction and execution overhead, and can distribute single-threaded code across lightweight cores. In these architectures, instead of predicting each single branch instruction, prediction is performed on a block-granularity using a *next block predictor* or *target predictor*. This predictor predicts the next block that will be fetched following the current block. As EDGE blocks can have multiple exits, each block can have multiple next block addresses depending on the history of the previously executed blocks and the execution path within the block determined by the predicates. As an example, Figure 8.1 shows a sample code, its dataflow representation and a diagram corresponding to the predicated dataflow block of the code. In the dataflow representation, the target fields of each instruction, represent a destination instruction and the type of the target. For example, p and $op1$ represent the *predicate* and *first operand* target types, respectively. The two branches in the original code (I_1 and I_3) are converted to dataflow test instructions (i_1 and i_3). During execution, once a test instruction executes, its predicate value (1 or 0) is sent to the consuming instructions of that test instruction. The small circles in the diagram indicate the predicate consumer instructions and their predicate polarity. The white and black circles indicate the instructions predicated on true and false, respectively. For instance, the `subi` only executes if the i_1 test instruction evaluates to zero. Depending on the value of the predicate instructions, this block takes one of three possible exits. If i_1 evaluates to 1, the next block will be block $B2$. If both i_1 and i_3 evaluate to 0, this block loops back to itself (block $B1$). Finally, if i_1 and i_3 evaluate to 0 and 1, this block branches to block $B3$. This model of predicated execution changes the control speculation problem from one-bit taken/not-taken prediction to multi-bit predicate path prediction when fetching each block. Thus, an accurate predictor for EDGE must use a global history of the predicates in previous blocks to predict the predicate path that will execute in the current block and then use that predicate path information to predict the next block. This section proposes the first such fast and accurate predictor called Iterative Path Predictor (IPP).

One drawback associated with predicated dataflow blocks is that the test instructions producing the predicates within blocks are executed and not predicted like normal branches. Our critical path analysis shows that when running SPEC benchmarks across 16 TFlex merged cores, on average about 50% of the critical cycles belong to instructions waiting for predicates. In Figure 8.1(c), i_1 will not execute until the value of $R1$ has arrived. Similarly, i_3 will not execute until both $R1$ and $R2$ have arrived and the result of the i_2 ($SUBI$) instruction is evaluated. To mitigate this execution bottleneck caused by intra-block predicates, IPP uses the predicted predicate path of each block to speculate on the value of predicates within that block, thus increasing the speculation rate among the distributed cores.

8.2.1 Integrated Predicate and Branch Predictor

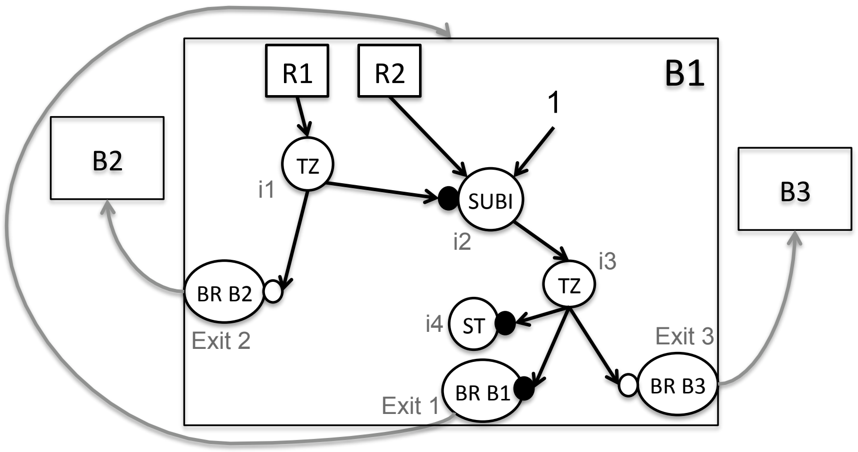
Previous EDGE microarchitectures predict the block exit in order to perform next block prediction. Figure 8.2(a) illustrates the block diagram of the next block predictor in each TFlex core. This 16K-bit predictor consists of two major components: (a) an exit predictor that is an Alpha 21264-like tournament predictor that predicts a three-bit exit code (the ISA allows between one and eight unique exits from each block) of the current block, and (b) a target predictor that uses the predicted exit code and the current block address to predict the next block address (PC). Because each exit can result from a different branch type, the target predictor supports various types of targets

```
I1:  bz R1, B2
I2:  subi a, R2, 1
I3:  bz a, B3
I4:  ST ADDR
I5:  j B1
```

(a) Initial representation

```
Read R1 <i1,op1>
Read R2 <i2,op1>
i1:  tz <e2,p><i2,p>
i2:  subi_f 1 <i3,op1>
i3:  tz <e1,p> <e3,p> <i4,p>
i4:  ST_f ADDR
e1:  br_f B1
e2:  br_t B2
e3:  br_t B3
```

(b) Dataflow representation



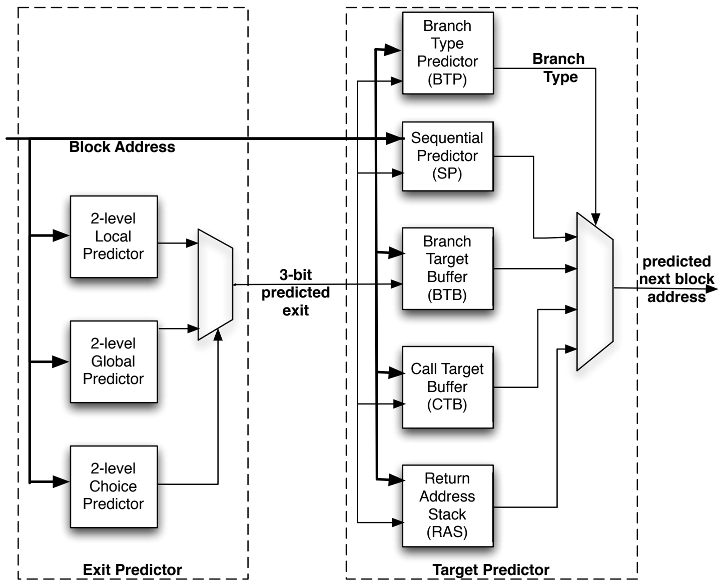
(c) Dataflow diagram

Figure 8.1: Sample code, its equivalent predicated dataflow representation, and the code diagram for the corresponding predicated dataflow block including two predicated execution paths and three possible exits.

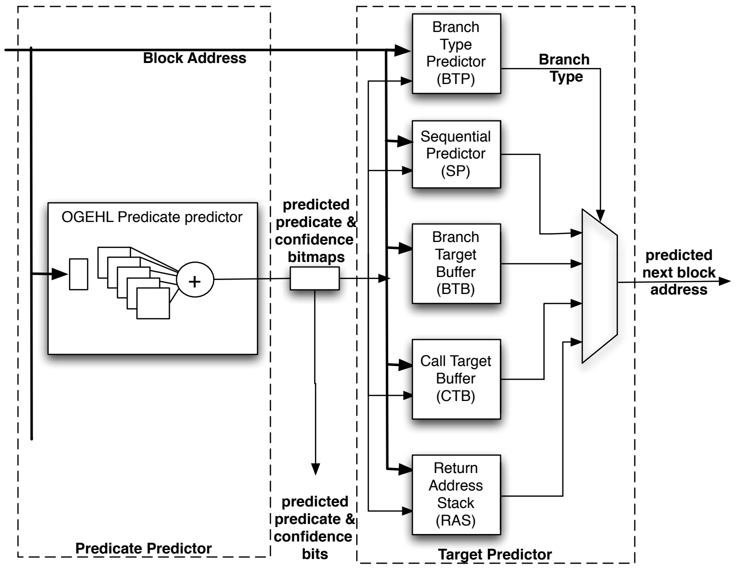
such as *sequential, branch, call*, and *return* targets. For the block shown in Figure 8.1(c), the TFlex exit predictor predicts which of the three exits from the block (*Exit* 1 to 3 in the figure) will be taken and then the target predictor maps the predicted exit value to one of the target block addresses ($B1$ to $B3$ in the figure).

Similar to the TFlex predictor, IPP is a fully distributed predictor with portions of prediction tables distributed across participating cores. Figure 8.2(b) shows the block diagram of the IPP predictor. Instead of predicting the exit code of the current block, IPP contains a predicate predictor that iteratively predicts the values of the predicates (predicate paths) in the current block. The predicted values are grouped together as a predicted *predicate bitmap* in which each bit represents a predicate in the block. For example, for the block shown in Figure 8.1(c), the bitmap will have two bits with the first and second bits predicting the results of the test instructions i_1 and i_3, respectively. The target predictor is similar to the target predictor used by the TFlex block predictor. It uses the predicted predicate bits (values) along with the block address to predict the target of the block. The rest of this subsection discusses the structure of the predicate predictor component in IPP.

Predicting predicates in each block is challenging since the number of dynamically executed predicates in each block is not known at prediction time. For simplicity, the predicate predictor used by IPP assumes a fixed number of predicates in each block. The predicate predictor component must predict multiple predicate values as quickly as possible so it will not become the



(a) TFlex next block predictor


Block Address
Branch Type Predictor (BTP)
Branch Type
Sequential Predictor (SP)
OGEHL Predicate predictor
predicted predicate & confidence bitmaps
Branch Target Buffer (BTB)
predicted next block address
Call Target Buffer (CTB)
Return Address Stack (RAS)
predicted predicate & confidence bits
Predicate Predictor
Target Predictor

(b) T3 iterative path predictor (IPP)

Figure 8.2: Block diagram of TFlex block predictor and T3 iterative path predictor.

bottleneck. After studying LTAGE [77], OGEHL [76] and other state-of-the-art predictors, we designed an optimized geometric history length (OGEHL) predictor [76] for predicate value (path) speculation. Figure 8.3 shows the original OGEHL branch predictor. The predictor predicts each branch in three steps. First, in the *hash compute* step, the branch address is hashed with the contents of the global history register (GHR) using multiple hash functions. Then, the produced hash values are used to index multiple prediction tables in the *table access* step. Each entry in these tables is a signed saturating counter. Finally, in the *prediction* step, the sum of the indexed counters in the prediction tables is calculated and its sign is used to perform prediction. Positive and negative correspond to taken and not-taken branches or true and false predicate values, respectivey. The absolute value of the sum is the estimated confidence level of the prediction. By comparing the confidence level to a threshold, a confidence bit is generated for each prediction. When the prediction is performed, the corresponding counters in the tables and the GHR value are updated speculatively. We use the best reported O-GEHL predictor in [76] with eight tables and a 200-bit global history register (modified from the original 125-bit GHR). Assuming this best-performing predictor distributed across 16 T3 merged cores, the size of the prediction tables stored on each core is about 8Kbits, which is equal to the size of the exit predictor in the original TFlex predictor shown in Figure 8.2(a). Therefore, using IPP does not incur any additional area overhead. To keep the global history registers consistent across cores, when a core performs a next block prediction, it broadcasts its

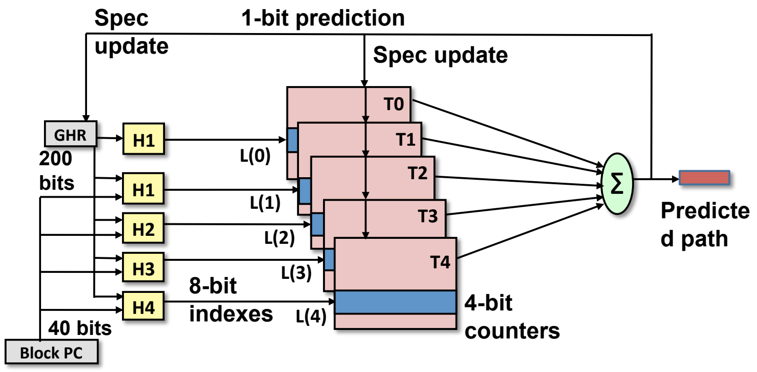
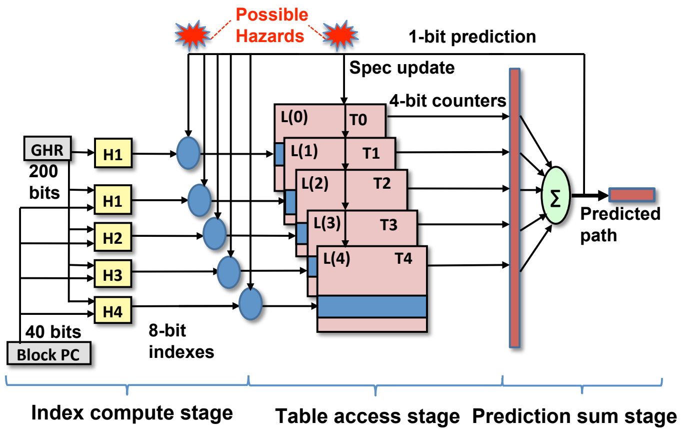


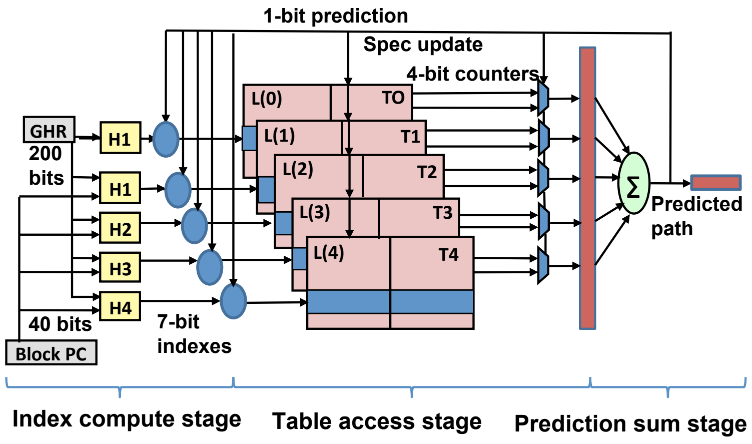
Figure 8.3: Basic OGEHL predictor.

changes to the GHR to other cores.

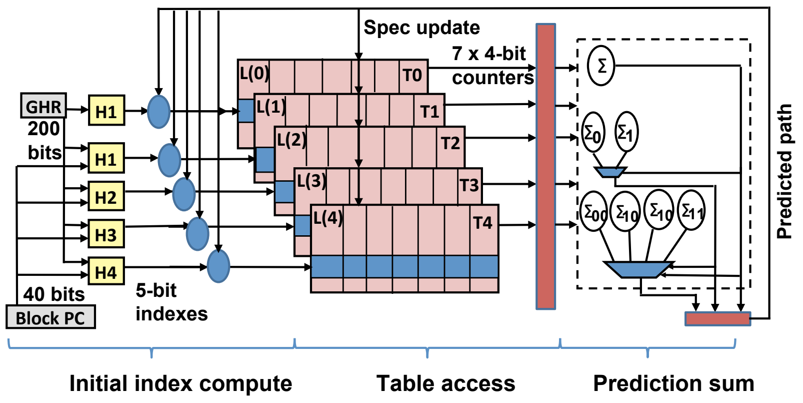
To accelerate the predicate path prediction, we optimize the OGEHL predictor by converting each step in the OGEGL predictor into a pipeline stage as shown in Figure 8.4(a). Although, this predictor can predict one predicate in each cycle, due to the speculative updates of GHR and prediction counters, there are possible data hazards in this pipeline when predicting back-to-back dependent predicates in one block. For example, if the second predicate in a block is $false$ only when the first predicate is $true$, this correlation is not captured in this pipeline because when the first prediction is still in flight, in the *prediction* stage, the second prediction is in the *access* stage. To address this issue, a hazard-free pipelined OGEHL shown in Figure 8.4(b) reads dual prediction values from each prediction table in the table access stage. The correct value is selected at the end of that stage depending on the prediction value computed in the *prediction* stage (selecting the second prediction based on the first prediction). Extending the same technique of reading dual predic-



(a) Pipelined OGEHL predictor



(b) Hazard-free pipelined OGEHL predictor



(c) Aggressive pipelined OGEHL predictor

Figure 8.4: Three OGEHL-based pipeline designed used for the T3 predicate predictor.

tions from the table, an aggressive pipelined predictor shown in Figure 8.4(c) reads dual predictions for multiple predicates within the same block in order to maximize prediction speed. In this predictor, seven counters are read from each table in the table access stage and then an adder tree is used to extract three prediction bits in the prediction stage. Such an aggressive predictor can perform three predictions in each cycle.

8.2.2 Speculative Execution of Predicate Paths

When the next target of a block is predicted, the predictor sends the predicted predicate bitmap to the core executing that block. It also sends another bitmap called a *confidence bitmap* with each bit representing the confidence of its corresponding predicted predicate. When an executing core receives the predication and confidence bitmaps, it stores the information required for speculative execution of the predicates in the instruction queue. The instruction queue is extended to contain one *confidence* bit and one *prediction* bit for each predicate-generating test instruction. For each predicate with its confidence bit set, the speculation starts immediately after receiving these bits by sending the predicted value to its destination instructions. For example, assume the bitmap associated with the block shown in Figure 8.1(c) is `00`, meaning that the i_1 and i_3 predicates are both predicted to be 0. In this case, the *store* instruction, i_4, is executed and block loops through $Exit1$ immediately, thus avoiding waiting for predicates to be computed and input registers R1 and R2 to arrive. If the bitmap is `10` or `11`, then $Exit2$ is immediately taken, thus ignoring all instructions in the block and branching directly to block $B2$.

For detecting predicate misspeculations, this mechanism relies on the dataflow execution model used by T3. The speculated test instructions in a block still receive their input values from other instructions inside the block. Once all inputs of such a speculated test instruction have arrived, that instruction executes as a normal instruction but does not send its output to its destination instructions again. Instead, the output of the test instruction is compared against the predicted value of that predicate and if the two do not match, a misspeculation flag is raised. Consequently, the block and all of the blocks that depend on it are flushed from the pipeline and the prediction tables are updated for that block.

8.3 Design Exploration Results

Table 8.1 compares different proposed pipelined IPP designs including the pipelined IPP, the hazard-free pipelined , and the aggressive pipelined IPP shown in Figures 8.4. In this experiment, each SPEC benchmark runs using 16 merged cores. This table presents MPKI (mispredictions per kilo instructions) for both next block prediction and predicate value speculation. It also presents speedups compared to the original TFlex predictor show in Figure 8.2(a). Using the basic pipelined IPP improves next block prediction MPKI from 4.03 to 3.29. By capturing the correlation between consecutive predicates in each block, the hazard-free pipeline improves MPKI to 2.93, while improving pred-

Table 8.1: Accuracy and speedups of different proposed IPP designs.

	TFlex next block predictor	Basic pipelined IPP	Hazard-free pipelined IPP	Aggressive pipelined IPP
Next block prediction MPKI	4.03	3.29	2.93	2.92
Predicate prediction MPKI	N/A	0.65	0.54	0.54
Average speedup	1.0	1.11	1.14	1.15

Table 8.2: Accuracy and speedups of the pipelined IPP when varying number of predicted predicates per block.

Number of predicted predicates per block	1	2	3	4	5
Next block prediction MPKI	4.43	4.00	2.86	2.93	2.96
Predicate prediction MPKI	0.10	0.29	0.44	0.54	0.57
Average speedup over TFlex	1.03	1.04	1.12	1.14	1.13

icate prediction MPKI from 0.65 down to 0.54. Of the 14% speedup achieved by the hazard-free IPP pipeline, the contributions of speculative execution of predicates and improved next block prediction accuracy are 12% and 2%, respectively. This predictor increases core-level energy consumption by 1.2%, most of which is consumed by the O-GEHL adders. However, energy saved by this predictor because of the improved next block and predicate prediction accuracy is about 6%, resulting in an overall energy improvement of 4.8%. The aggressive pipelined design results in slightly higher speedup compared to the hazard-free pipeline, which considering the design overhead and complexity of both designs, the hazard-free pipeline is a preferred candidate for T3.

Table 8.2 evaluates the hazard-free IPP design when varying the number of predicted predicate values per block. The next block prediction accuracy first improves when increasing predicted branches (predicate values) from 1 to 3 and then degrades. This observation is supported by the fact that for most SPEC benchmarks, the average number of executed predicates per block is three. The predicate prediction MPKI, however, increases consistently as the number of speculated predicates increases from 1 to 5. However, these MPKIs are very low and do not highly affect performance. Although the best next block prediction is achieved when predicting three predicates per block, the best speedup is achieved when predicting 4 predicates per block due to the increased intra-block speculation. This speedup is only slightly better than the speedup achieved when predicting 3 predicates per block. So, considering the increase in the next block prediction misses when predicting 4 predicates, predicting three predicates per block can potentially result in higher power efficiency.

8.4 Summary

This chapter proposes a combined branch and predicate predictor for T3 called Iterative Path Predictor. This predictor solves the low multi-exit next bock prediction accuracy issue and low speculation rate issue caused by heavy predicate execution. The predictor predicts a multi-bit predicate path within each block and uses it to accurately predict the next block following that block. In order to accelerate the prediction of intra-block predicate path, IPP employs a novel pipelined OGEHL predictor with no area overhead. The predicted predicate path is used for both improving next block predictor accuracy and speculative execution of predicates. This predictor does not incur any area overheads while saving 14% execution delay and 5% core-wide energy with 16 composed T3 cores.

Chapter 9

Efficient Operand Delivery using Exposed Operand Broadcasts

9.1 Introduction

Communicating operands between instructions is a major source of energy consumption in modern processors. A wide variety of operand communication mechanisms have been employed by different architectures. For example in superscalar processors, to wake up all consumer instructions of a completing instruction, physical register tags are broadcast to power-hungry Content Addressable Memories (CAMs), and operands are obtained from a complex bypass network or by a register file with many ports. A mechanism commonly used for operand communication in dataflow architectures such as EDGE designs, is point-to-point communication. Dataflow is highly efficient when a producing instruction has a single consumer; the operand is directly routed to the consumer, often just requiring a random-access write into the consumer's reservation station. If the producer has many consumers, however, dataflow implementations typically build an inefficient software fanout tree of operand-propagating instructions (that we call *move* instructions). These two mechanisms are efficient under different scenarios: broadcasts should be used when there are many consumers currently in flight (meaning they are in the instruction window), dataflow should be used when there are few consumers, and registers should be used to hold values when the consumers are not yet present in the instruction window.

The chapter discusses and evaluates a compiler-assisted hybrid instruction communication mechanism proposed by Li et al. [48, 69] called Exposed Operand Broadcasts (EOBs). This mechanism augments EDGE dataflow instruction communication model with a small number of architecturally exposed broadcasts within the instruction window. A narrow CAM uses high-fanout instructions to send their operands to their multiple consumers, but only unissued instructions waiting for an architecturally specified broadcast actually perform the CAM matches. The other instructions in the instruction window do not participate in the tag matching, thus saving energy. All other instructions, which have low-fanout, rely on the point-to-point token communication model. The determination of which instructions use tokens and which use broadcasts is made statically by the compiler and is communicated to the hardware via the modified ISA. As a result, this method does not require instruction dependence detection and instruction categorization at runtime. In addition, the compiler can reduce the bit width of the needed CAM by efficiently reusing the tags for non-overlapping live range broadcasts. The rest of the chapter gives an overview on EOBs and their required compiler analysis, and then discusses how their support is added to T3, and finally presents a design exploration of the EOBs integrated in T3.

9.2 Dataflow Fanout Overhead

By eliminating register renaming, result broadcast, and associative tag matching in the instruction queue, the direct dataflow intra-block communication achieves major energy savings for low-fanout operands compared to conventional out-of-order designs. However, the energy savings are limited in the case of high-fanout instructions for which the compiler needs to generate software fanout trees [32]. Each instruction in the EDGE ISA can encode up to two destinations. As a result, if an instruction has a fanout of more than two, the compiler inserts two- or three-target *move* instructions to form a dataflow fanout tree for operand delivery. Previous work [32] has shown that for the SPEC benchmarks, 25% of all instructions are *move* instructions. These fanout *move* trees manifest themselves at runtime in the form of extra power consumption and execution delay. To alleviate this issue, this dissertation proposes a novel *hybrid* operand delivery that exploits compile-time analysis to minimize both the delay and energy overhead of operand delivery within each distributed T3 core. This mechanism uses direct dataflow communication for low-fanout operands and compiler-generated ISA-exposed operand broadcasts (EOBs) for high-fanout operands. These limited EOBs eliminate almost all of the fanout overhead of the move instructions. Move instruction removal results in fetch and execution of fewer instructions, fewer blocks (through more efficient block formation), and large energy savings.

9.2.1 EOB Assignment and Instruction Encoding

This subsection briefly discusses EOBs and how the compiler generates them. For a more detailed discussion, please refer to [48, 69]. The original EDGE compiler [78] generates blocks containing instructions in dataflow format in which each instruction directly specifies each of its consumers using a 7-bit instruction identifier. As shown in Figure 9.1, each instruction can encode up to two target instructions in the same block. The *xop* field in this figure is an extended opcode that contains multiple unused bits. During block formation, the compiler identifies and marks the instructions that have more than two targets. Later, the compiler adds *move* fanout trees for those high-fanout instructions during the code generation phase.

The modified EOB-enabled compiler accomplishes two additional tasks, choosing which high-fanout instructions should be selected for one of the limited intra-block broadcasts, and assigning one of the static EOBs to each selected instruction. The compiler uses a greedy algorithm, sorting all instructions in a block with more than two targets and selecting those instructions based on the number of targets. Starting from the beginning of the list, the compiler assigns each instruction in the list an EOB from fixed number of available EOBs. The number of available EOBs is determined by a microarchitectural parameter called $MaxEOB$. The send and receive EOBs must be encoded in both operand broadcast producing and consuming instructions.

Figure 9.2 illustrates a sample program, its equivalent dataflow representation, and its equivalent hybrid dataflow/EOB representation generated

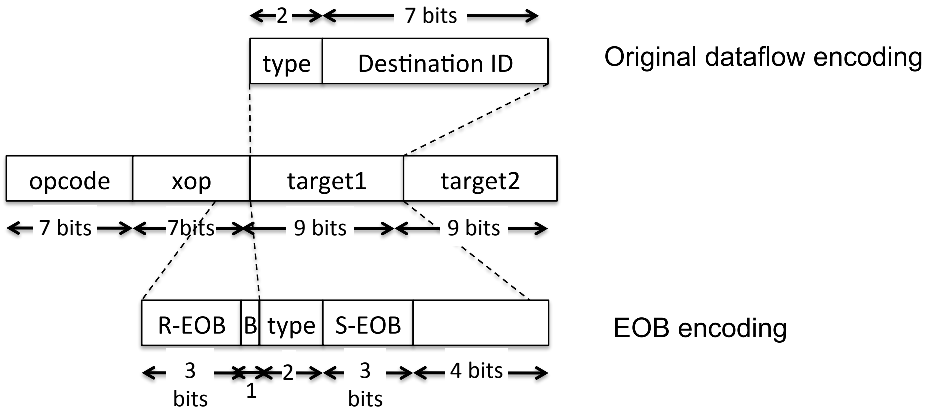


Figure 9.1: T3 instruction encoding with support for EOBs. S, R, and B refer to Send, Receive, and Broadcast enable, respectively.

by the modified compiler. In Figure 9.2(a), *a, b, d, g* and x are the inputs read from registers and except for *stores*, the first operand of each instruction is the destination. In the dataflow code shown in Figure 9.2(b), instruction i_1 only encodes two of its three targets. Therefore, the compiler inserts a *move* instruction, instruction i_{1a}, to generate the fanout tree for that instruction. For the hybrid communication model shown in Figure 9.2(c), the compiler assigns an EOB (1 in this example) to i_1, the instruction with high fanout, and encodes the broadcast information into both i_1 and its consuming instructions (instructions i_2, i_3 and i_5). Finally, the compiler uses dataflow direct communiction for the remaining low-fanout instructions, e.g. instruction i_2 in Figure 9.2. The branch instruction i_4 in the original code is converted to a predicate using a test instruction, i_4 in the dataflow code. The two store instructions are on the true and false predicated paths of this predicate.

```
I1:     add c, a, b
I2:     sub e, c, d
I3:     add f, c, g
I4:     bz x L1
I5:     st c, f
I5a:    j EXIT
L1:
I6:     st e, f
```

(a) Initial representation

```
i1:     add <i2, op1> <i1a, op1>
i1a:    mov <i3, op1> <i5 op1>
i2:     sub <i6, op1>
i3:     add <i5, op2> <i6, op2>
i4:     testnz <i5, pred><i6, pred>
i5:     st_t
i6:     st_f
```

(b) Dataflow representation

```
i1:     add [S-EOB=1, op1]
i2:     sub [R-EOB=1] <i6, op1>
i3:     add [R-EOB=1] <i5, op2><i6, op2>
i4:     testnz <i5, pred><i6, pred>
i5:     st_t [R-EOB=1]
i6:     st_f
```

(c) Hybrid dataflow/EOB representation

Figure 9.2: A sample code and corresponding code conversions in the modified compiler for the hybrid dataflow/EOB model.

9.2.2 Microarchitectural Support for EOBs

This subsection discusses how EOBs are integrated into the T3 system. To implement EOBs in T3 cores, a small *EOB CAM* array stores the receive EOBs of broadcast receiver instructions in the instruction queue. Figure 9.3 illustrates the instruction queue of a single T3 core when running the broadcast instruction i_1 in the sample code shown in Figure 9.2(c). When the broadcast instruction executes, its send EOB (value *001* in this example) is sent to be compared against all the potential broadcast receiver instructions in the instruction queue. Only a subset of instructions in the instruction queue are broadcast receivers, while the rest need no EOB comparison. Operands that have already received their broadcast do not have to perform CAM matches, saving further energy. Upon an *EOB CAM* match, the hardware generates a write-enable signal to write the operand into the instruction queue entry of the corresponding receiver instruction. The broadcast type field of the sender instruction ($operand1$ in this example) is used to select the column corresponding to the receivers. Tag delivery and operand delivery do not happen on the same cycle. Similar to superscalar operand delivery networks, the EOB of the executing sender instruction is first delivered one cycle before instruction execution completes. On the next cycle, when the result of the broadcast instruction is ready, its output is written simultaneously into all matching operand buffers in the instruction window.

Figure 9.3 also illustrates a sample circuit implementation for the compare logic in each *EOB CAM* entry. The CAM tag size in this figure is three

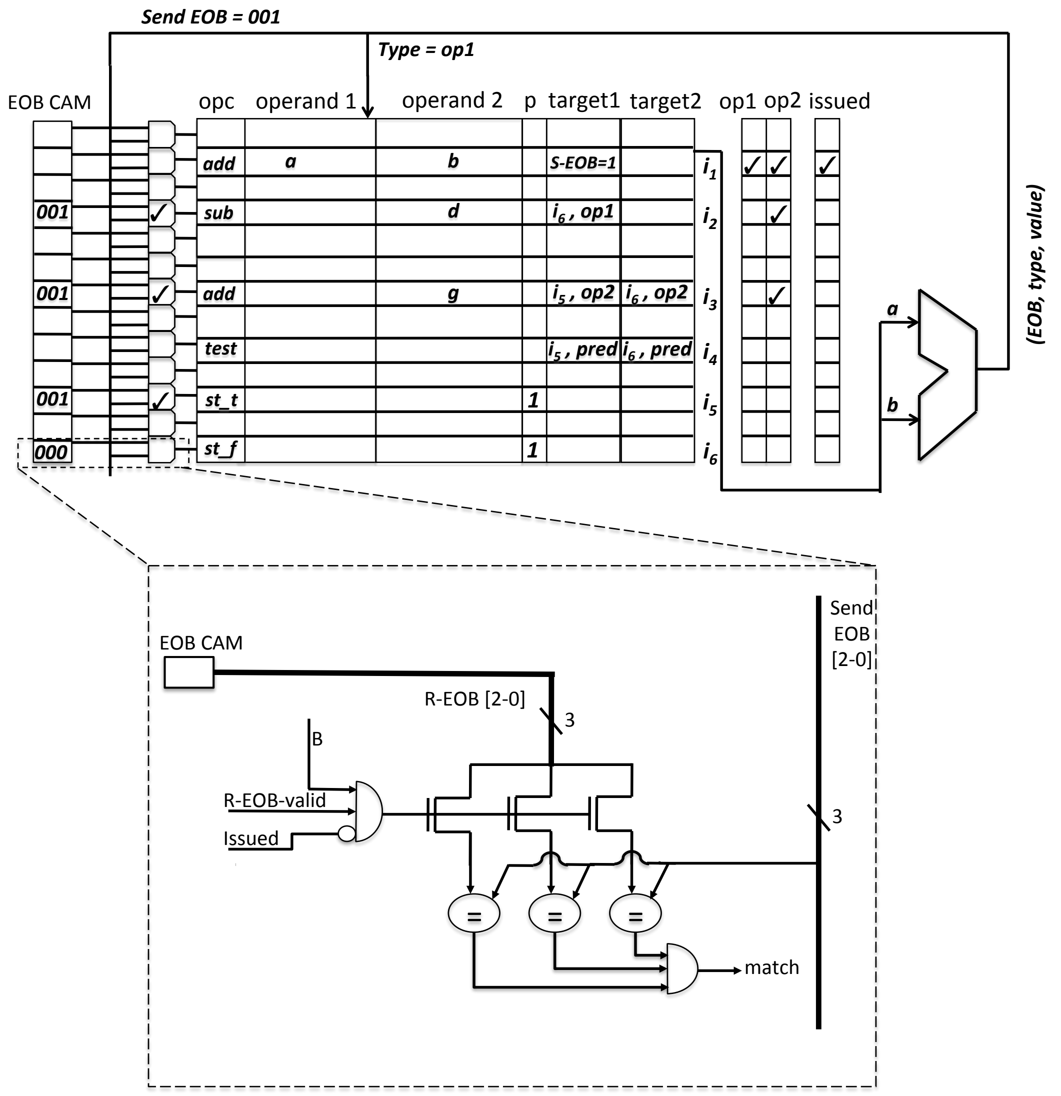


Figure 9.3: Execution of a broadcast instruction in the IQ (top) and the compare logic for each EOB CAM (bottom).

bits which represents the bit width of $EOBs$. In this circuit, the compare logic is disabled if one of the following conditions is true: (1) if the instruction corresponding to the CAM entry has been previously issued, (2) if the receive EOB of the instruction corresponding to the CAM entry is not valid, which means the instruction is not a broadcast receiver (for example instruction i_5 in Figures 9.2 and 9.3), or (3) if the executed instruction is not a broadcast sender. Despite the fact that they both use CAMs, EOBs are more energy efficient than the instruction communication model in superscalar processors for several reasons. First, because EOBs use small identifiers, the bit width of the CAM is small compared to a superscalar design which must track a larger number of renameable physical registers. Second, the compiler can select which instruction operands are broadcast, which in practice is a small fraction of the total instruction count. Third, only a portion of instructions in the queue are broadcast receivers and perform an EOB comparison during each broadcast.

9.3 Design Space Exploration Results

Increasing the number of the available EOBs ($MaxEOBs$) from zero to 128 (the maximum number of instructions in each EDGE block) produces fewer fanout trees and adds more broadcasts to the code. By choosing an appropriate value for this ISA parameter, the compiler is able to minimize total energy consumed by fanout trees and EOBs, while achieving a decent speedup as a result of using EOBs for high-fanout instructions. Figure 9.4 illustrates

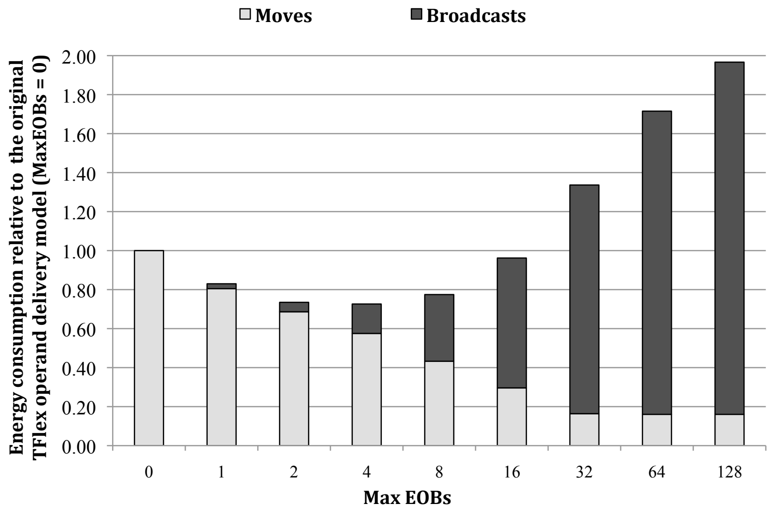


Figure 9.4: Averaged energy breakdown between move instructions and broadcasts for various numbers of available EOBs for SPEC benchmarks.

the energy breakdown into executed *move* and broadcast instructions for a variety of $MaxEOBs$ values on the SPEC benchmarks each running across 16 merged cores. The energy models used to generate this graphs are derived from the validated TRIPS energy models. Chapter 10 discusses our detailed and accurate energy modeling methodology. The energy values are normalized to the total energy consumed by *move* instructions when instructions within each block communicate only using dataflow ($MaxEOBs = 0$). When only using dataflow (the original TFlex operand delivery), all operand delivery energy overheads are caused by the *move* instructions. Allowing one or two broadcast operations in each block, $MaxEOBs$ of 1 and 2, we observe a sharp reduction in the energy consumed by *move* instructions. The compiler chooses the instructions with highest fanout first when assigning EOBs. For these

$MaxEOBs$ values, the energy consumed by EOBs is very low. As we increase the total number of EOBs, the energy consumed by broadcast operations increases dramatically and fewer move instructions are removed. At some point, the broadcast energy becomes dominant. For high numbers of $MaxEOBs$, the broadcast energy is an order of magnitude larger than the energy consumed by *move* instructions. The key observation in this graph is that allowing only 4 to 8 broadcasts in each block minimizes the total energy consumed by *moves* and broadcasts. For such $MaxEOBs$, the total energy is about 28% lower than the energy consumed by the baseline TFlex ($MaxEOBs = 0$) and about 2.7x lower than when $MaxEOBs$ is equal to 128. These results show that the compiler is able to achieve a better trade-off in terms of power breakdown by selecting a critical subset of high-fanout instructions in each block. We also note that for $MaxEOBs$ larger than 32, the energy consumed by *move* instructions is at a minimum and does not change, but the *EOB CAM* becomes wider so the energy consumed by EOBs continues growing.

Using 3-bit EOBs removes 73% of dataflow fanout instructions and instead 8% of all instructions are encoded as the EOB senders. These instructions send EOBs to 34% of instructions (EOB receivers). Using 3-bit EOBs results in about 10% total energy reduction on T3 cores. The consumed energy is reduced in two ways: (1) it saves the energy consumed during execution of the fan-out trees which constituent more than 24% of all instructions; and (2) by better utilizing the instruction blocks, it reduces the fetch and decode operations by executing 5% fewer blocks.

9.4 Summary

This chapter discusses a compiler-assisted hybrid operand delivery mechanism proposed by Li et al. [48] and explains how it is integrated into T3. Instead of using dynamic hardware-based pointer chasing, this mechanism relies on the compiler to categorize instructions for token or broadcast operations. In this model, the compiler assigns broadcasts for critical operands that had many consumers, and dataflow for the rest of operands. The compiler analyzes the program to select the best operand communication mechanism for each instruction. At the same time, the block-atomic EDGE model made it simple to perform that analysis in the compiler, and allocate a number of architecturally exposed broadcasts to each instruction block. Furthermore, compiler performs complex optimizations without hardware cost and execution-time penalty like the dynamic approaches. By limiting the number of broadcasts, the CAMs searching for broadcast IDs can be kept narrow, and only those instructions that have not yet issued and that actually need a broadcast operand need to be performing CAM matches. Exploiting both low-overhead architecturally exposed broadcasts and direct dataflow communication, T3 supports fast and energy-efficient operand delivery for high- and low-fanout instructions. By using 3-bit wide EOBs, the overall operand delivery overhead is minimize, which translates to 10% core-wide energy reduction and 5% increase in performance when running across 16 composed cores.

Chapter 10

Integrated T3 Power and Performance Results

This chapter first presents a power/performance evaluation of all T3 components across a range of core counts. After discussing our methodology for power and performance analysis, the chapter then compares the fully integrated T3 system to previous EDGE microarchitectures (TRIPS and TFlex) that have different core composition granularities and microarchitectural features. To illustrate power/performance tradeoffs achieved by T3, the chapter then compares the performance/power flexibility of the T3 microarchitecture against several design points in the performance and power spectrum of production processors such as Intel Atom and Core 2 processors. Finally, the chapter presents the final critical path analysis of T3 in order to find the last remaining bottlenecks and possible opportunities for further improvements.

10.1 Methodology

We use an execution-driven, cycle-accurate simulator to simulate the TRIPS, TFlex, and T3 processors [41]. The simulator is validated against the cycles collected from the TRIPS prototype chip. In TFlex or T3 modes, the simulator supports different configurations in which a single thread can run across a number of cores ranging from 1 to 16 cores in powers of 2. We limit the number of merged cores between 1 and 16 as performance and power scaling does not improve much when merging more than 16 cores. The power model uses CACTI [87] models for all major structures such as instruction and data caches, SRAM arrays, register arrays, branch predictor tables, load-store queue CAMs, and on-chip network router FIFOs to obtain a per-access energy for each structure. Combined with access counts from the architectural simulator, these per-access energies provide the energy dissipated in these structures. The power models for integer and floating point ALUs are derived from both Wattch [13] and the TRIPS hardware design database. The combinational logic power in various microarchitectural units is modeled based on detailed gate and parasitic capacitances extracted from RTL models and activity factor estimates from the simulator. The baseline EDGE power models at 130nm are suitably scaled down to 45nm using linear technology scaling. For a complete review of out power modeling methodology, please refer to [34, 35].

We use a supply voltage of 1.1 Volts and a core frequency of 2.4 GHz for the TRIPS, TFlex, and T3 platforms. Our benchmarks include 15 SPEC 2000 [2] benchmarks (7 integer and 8 floating point) each simulated with a single simpoint region of 100 million instructions (the Fortran and non-compilable SPEC benchmarks are excluded).

We accurately model the delay of each optimization used by the T3 simulator. Also, we use CACTI and scaled TRIPS power models to estimate the power consumed by the tables or combinational logics used by various

Table 10.1: T3 optimizations.

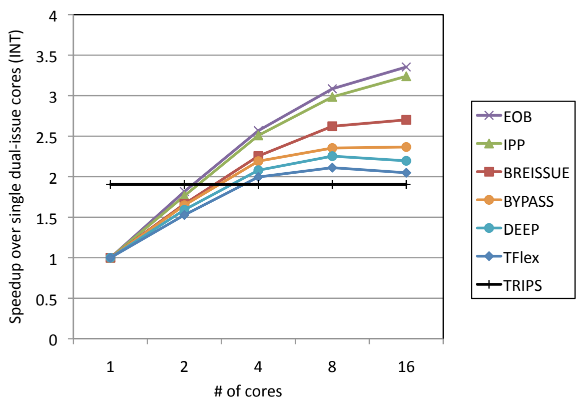
Optimization	Configuration
EOB	Each core supports hybrid communication explained in Chapter 9 using 3-bit EOBs ($MaxEOB$ equal to eight) + IPP.
IPP	Instead of block predictor in [41], each core uses the hazard-free pipelined IPP and predicts 4 predicates per block explained in Chapter 8 + BYPASS.
BYPASS	Enabling last-arriving register bypass [68] from producer cores to consumers (explained in Chapter 6) + BREISSUE.
BREISSUE	Enabling block reissue mechanism [68] explained in Chapter 7 + DEEP.
DEEP	Using the deep block mapping [67]explained in Chapter 5, in which all instructions in each block are mapped to just one core.

T3 features, such as the O-GEHL tables used by IPP or the EOB CAM and comparators.

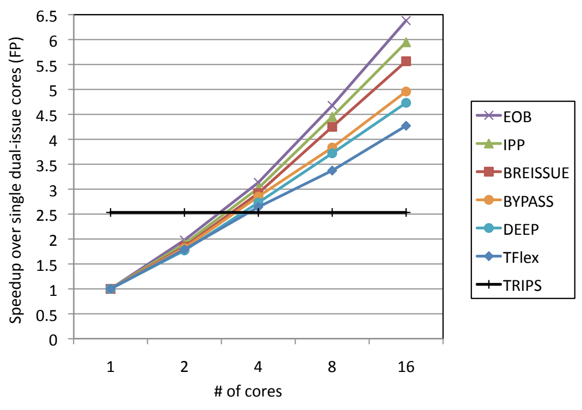
Table 10.1 lists the optimization mechanisms that we model for the integrated T3 processor. The EOBs used in these experiments are 3 bits wide, IPP uses the hazard-free pipeline predicting up to 4 predicates per blocks.

10.2 Performance and Energy Scalability Results

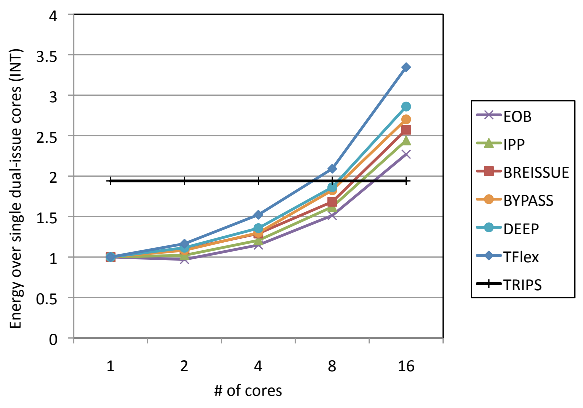
Figure 10.1 shows the average speedup, energy consumption (L2 energy excluded), and inverse energy-delay-product for TRIPS, TFlex, and T3 configurations. These graphs are normalized against runs on a single TFlex core. The T3 experiments are inclusive meaning that each experiment includes



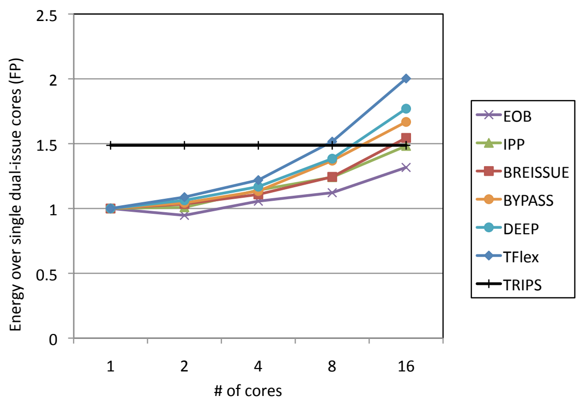
(a) SPEC INT Speedup



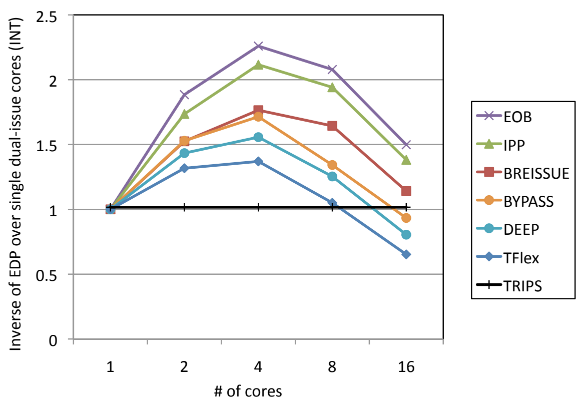
(b) SPEC FP Speedup



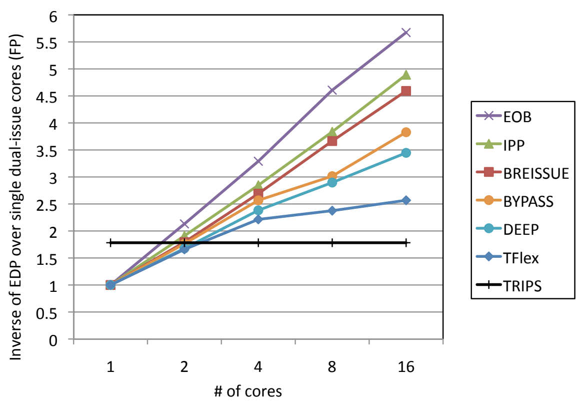
(c) SPEC INT Energy



(d) SPEC FP Energy



(e) SPEC INT Inverse Energy-delay-product



(f) SPEC FP Inverse Energy-delay-product

Figure 10.1: Average speedups, energy, and inverse of energy-delay-product over single core for the SPEC benchmarks with varying the numbers of merged cores and optimization mechanisms.

features added by all its previous experiments. For example, DEEP only includes the deep mapping model but EOB represents the complete integrated T3 system including deep mapping, register bypass, block reissue, iterative path prediction, and dataflow/EOB communication. In these graphs, T3 and TFlex charts are reported in different configurations each running different core counts ranging from 1 to 16. TRIPS results are straight lines as that microarchitecture does not support composability.

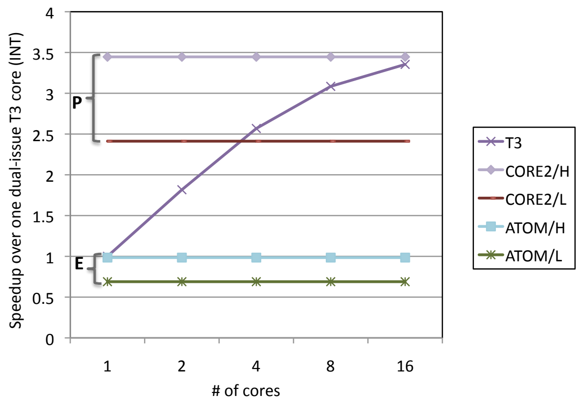
For INT benchmarks, Figures 10.1(a) and 10.1(c) show that *TFlex-8* (TFlex using 8 cores) outperforms TRIPS by about 1.12× while consuming slightly more energy. However, relying on the optimized microarchitectural components, *T3-8* (EOB charts across 8 cores in the figure), significantly outperforms TRIPS by 1.43× while consuming about 25% less energy. This significant simultaneous reduction in consumed energy and increase in performance of the T3 system translates to a major increase in energy efficiency, which is mostly attributed to the IPP and EOBs. *T3-4* achieves the best inverse-energy-delay-product (EDP) as shown in Figure 10.1(a). This value is 1.8× of that of *TFlex-4* EDP with more than half of this increase caused by the combination of IPP and EOBs. For FP benchmarks, *TFlex-16* outperforms TRIPS by about 1.7× while consuming 30% more energy. *T3-16* (EOB charts), on the other hand, outperforms TRIPS by about 2.5× while consuming 1.1× less energy. *T3-16* reaches the best inverse-EDP and inverse-ED^{2}P which are 2.6× and 7× better than those of TRIPS.

To better quantify power and performance benefits of the features proposed in this dissertation for the T3 system, we focus on the speed and power breakdown for INT benchmarks, which are inherently hard for a compiler to parallelize automatically. On average, *T3-16* outperforms *TFlex-16* by about 1.5× across both INT and FP benchmarks, which translates to a speedup of about 50%. For the INT benchmarks, the speedups stem primarily from the IPP (14%), deep block mapping (7%), and block reissue (11%). As shown in the energy graphs, the T3 optimized cores save significant energy compared to the TFlex. For example *T3-16* consumes about 38% less energy than *TFlex-16* for SPEC INT benchmarks. The main energy savers are EOBs (10%), deep block mapping (8%), and block reissue (7%). These energy savings come from (a) reduction in executed blocks and fanout move instructions as a result of using EOBs, (b) skipped fetch and decode operations as a result of reissuing blocks as they are still in the window, and (c) the reduction in cross-chip communication as a result of localizing intra-block communication within cores.

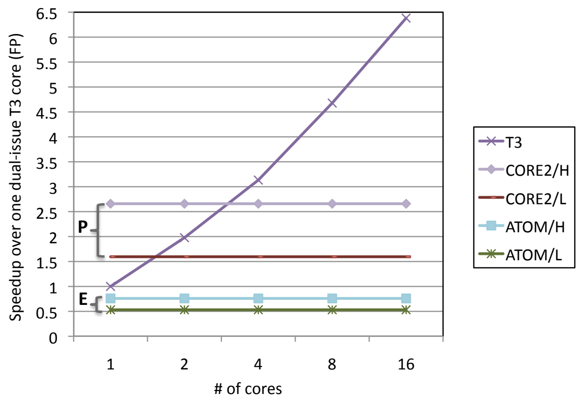
10.3 T3 Power Performance Tradeoffs

To examine the performance/power flexibility of the T3 microarchitecture, we compare it to several design points in the performance and power spectrum of production processors. An exact comparison is extremely challenging because publicly-available validated x86 power and performance models do not exist. We use the Intel Core 2 and Atom as representatives for high performance and lower power platforms respectively, and rely on the chip power and performance measurement results reported in [25] for these platforms with the technology node identical to T3, 45nm. We use the McPAT [49] models to estimate the core power consumption to compare against T3. We choose Core 2 and Atom as their power breakdowns are close to the high-performance and low-energy cores presented supported by McPAT [49]. The main idea of such a comparison is not a detailed, head-to-head comparison of T3 to these platforms, but to demonstrate the power/performance flexibility offered by T3 in the context of such platforms. While we recognize that our methodology is not ideal, we believe it has sufficient fidelity to demonstrate the potential of *one* T3 processor that operates on a wide spectrum of power/performance regions covered by a number of commercial products.

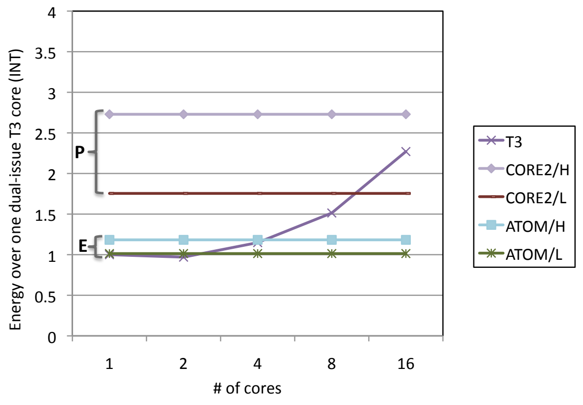
Figure 10.2 reports relative performance, energy and inverse-EDP results of various platforms. In each graph, different voltage and frequency operating points of Core 2 represent high-performance operating region (marked P). Similarly, operating points of Atom represents the low-energy operating region (marked E). Table 10.2 summarizes the operating points of different platforms in this experiment, which are extracted from [1, 25]. T3 runs only vary the number of composed cores with a fixed frequency and voltage equal to that of the the CORE2/H operating point. As shown in Figure 10.2, T3 achieves high energy efficiency in both low-energy and high-performance regions. By fusing a few of these T3 optimized cores, we can achieve major performance boosts in low-energy regimes. For example, while the energy consumed by *T3-2* falls within the low-energy region (Figures 10.2(c) and 10.2(d)), its performance is close to the range of the high-performance region (Figures 10.2(a)



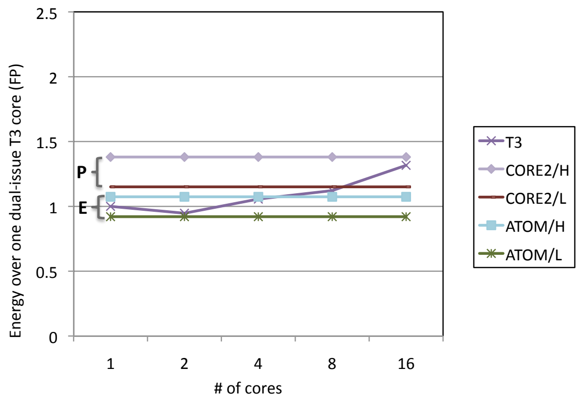
(a) SPEC INT Speedup



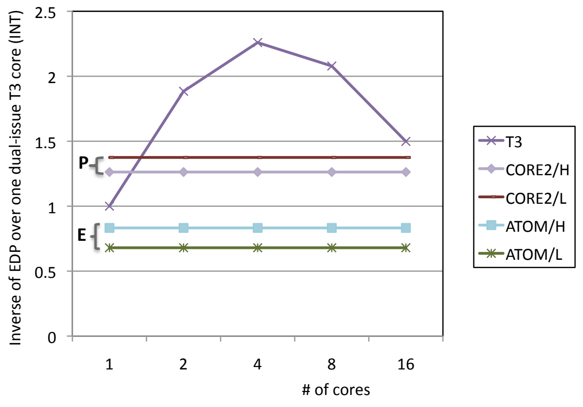
(b) SPEC FP Speedup



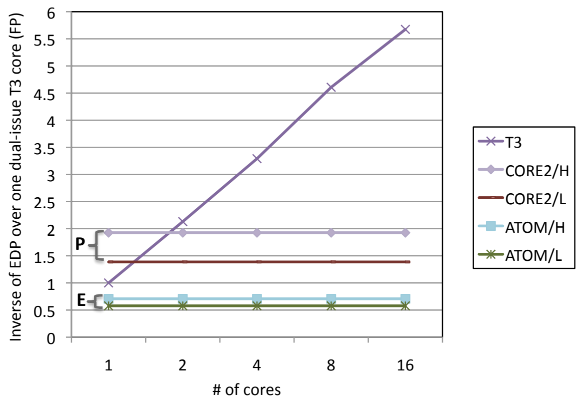
(c) SPEC INT Energy



(d) SPEC FP Energy



(e) SPEC INT Inverse Energy-delay-product



(f) SPEC FP Inverse Energy-delay-product

Figure 10.2: Average speedups, energy, and inverse-EDP over single core with varying the numbers of merged cores.

Table 10.2: Configurations for T3, Core2 and Atom platforms [1, 25].

	T3	CORE2/H	CORE2/L	ATOM/H	ATOM/L
Vdd (volts)	1.1	1.1	0.9	1.1	0.8
Frequency	2.4GHz	2.4GHz	1.6GHz	1.6GHz	800MHz

and 10.2(b)). On the other hand, merging more cores significantly boosts performance at a relatively small energy cost. For example, while *T3-4* and *T3-8* perform in or above the high-performance region, their consumed energy is below this region.

Finally, T3 not only performs in these energy/performance regions, but also covers a much larger space of operating points, which is covered partially by the Core 2 and Atom processors in this case study, thus extending the range of power/performance trade-offs beyond DVFS on conventional processors. This degree of energy efficiency and flexibility in T3 is an independent feature in addition to DVFS. T3 can combine this feature with DVFS to further extend the range of power/performance trade-offs. For instance, 1, 2, 4, 8 or 16 composed cores with 5 DVFS points provides 25 different highly energy-efficient operating points in the power/performance spectrum as opposed to just 5 via DVFS alone.

10.4 Final Bottleneck Analysis

Figure 10.3 compares the critical path pattern for flat mapping, deep mapping, register bypassing, block reissues and IPP proposed in this dissertation when applied cumulatively. Each bar is labeled by the optimization mechanism added the previous set to the left. The right most bar in this graph shows the results for the last step of our analysis including all four used optimizations for the 16-core configuration normalized against the 1-core results. We do not include the results for EOBs as they are very similar to the ones for IPP. The only difference is that the contribution of the *instruction execution* component will be slightly reduced as a result of elimination of the fanout *move* trees.

Applying IPP reduces the criticality of *instruction execute*, *register reads* and *data cache misses*. This is mostly the effect of intra-block speculation enabled by IPP as predicate values are predicted, instead of being evaluated in the execution stage. Whereas without this feature, before getting executed, the intra-block predicates have to wait for memory or register input values used in computing those predicates. Consequently, IPP reduces the contribution of register reads and memory accesses on the critical path.

The combination of different mechanisms proposed for T3 in this dissertation almost eliminates *cross-core communication*, *fetch stalls predicate execution* and *move fanout trees* from the critical path. At this point, *Instruction execution* is the only dominant critical resource in T3. This includes mostly the time spent on executing instructions in the functional units. As a result, the order of instructions in each T3 core can affect the delay associated with this resource. The current spatial path scheduling SPS [20] for placing instructions in the instruction queue of each core operates based on the statically

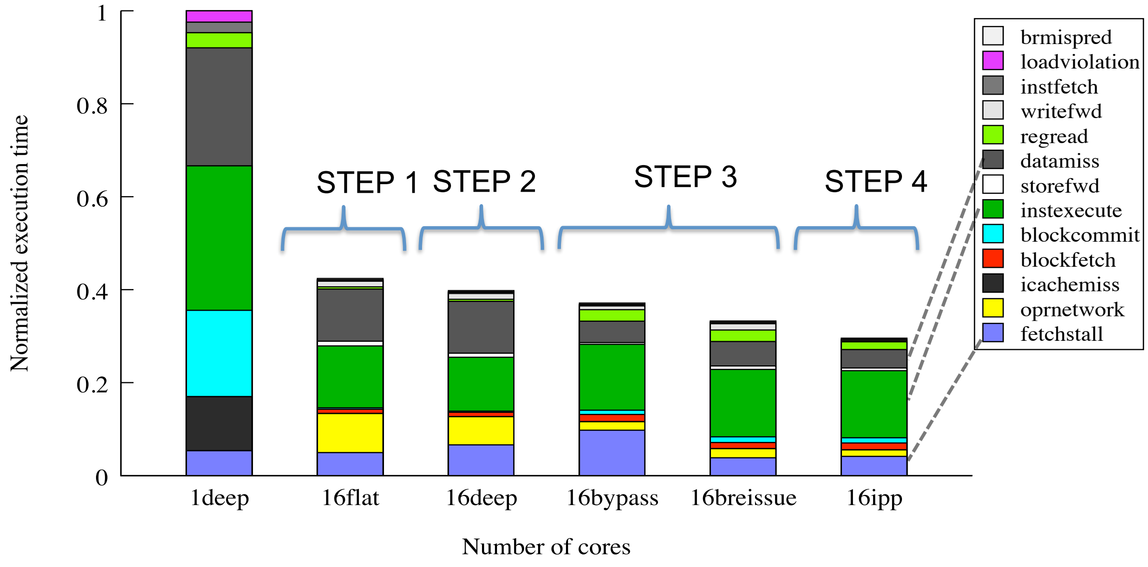


Figure 10.3: System-level breakdown when applying deep block mapping, register bypassing, block reissue and iterative path predictor.

estimated critical path within each block. This algorithm does not consider the predicate prediction enabled using IPP proposed in Chapter 8. When the predicates are predicted, the critical path within the block changes from the critical path of the baseline TFlex in which the predicates are evaluated. With predicate prediction enabled, register reads or loads leading to predicate test instructions are no longer on the critical if that test instruction is critical. Therefore, a revision of the SPS scheduling algorithm for changing the issue order (criticality) of instructions of each block in the instruction queue of the core executing that block, can further improve performance.

10.5 Summary

Exploiting novel mechanisms, T3 demonstrates significant performance and energy advantages over previous composable EDGE architectures. We compare the performance and energy efficiency of T3 against previous EDGE architectures. On SPEC CINT2000, T3 increases average performance appreciably (over 47% with eight composed cores) while simultaneously reducing the energy consumed (27% with eight cores), which translates to about 2x improved energy delay product, as compared to TFlex. Furthermore, T3 achieves high energy efficiency at different power and performance operating points across a wide power/performance spectrum.

We examine the performance/power flexibility of T3 by comparing it to real conventional platforms by using both hardware measurements [25] and analytical power models [49]. For high-performance (10∼30 watts range) and low-energy references (1∼3 watts range), we use an Intel Core 2 and an Intel Atom processors, respectively. With low core counts (one or two), T3 consumed energy is in the low-energy region while performing close to the high-performance region. When running with four or more composed cores per thread, T3 improves performance significantly while its consumed energy is below the energy ranges of the high-performance region. This degree of flexibility and energy efficiency allows T3 to explore power/performance trade-offs beyond those of conventional processors.

Our final critical path analysis shows that all dominant bottlenecks are almost eliminated using the proposed mechanisms for T3 in this dissertation.

The only dominant critical resource is the time spent in ALUs for executing instructions. This resource can be further sped up by employing an instruction scheduling algorithm rearranging the issue order of instructions considering the T3 predicate prediction feature enabled by IPP.

Chapter 11

Conclusions

As voltage scaling diminishes, processors need to rely on scalable architectural innovations to operate at different energy/performance operating points while maximizing energy efficiency at each point. Composable architectures can span a wide range of energy/performance operating points by enabling multiple simple cores to compose a larger and more powerful core. Explicit Data Graph Execution (EDGE) architectures represent a highly scalable class of composable processors that exploit predicated dataflow block execution and distributed microarchitectures. However, prior EDGE architectures suffer from major energy and performance bottlenecks. This dissertation studies these bottlenecks and proposes a new EDGE architecture called T3 [69], addressing these issues and fulfilling the original composition and power efficiency promises of EDGE. To conclude this dissertation, we discuss the bottlenecks reported in this study and the methods T3 employs to addresses those bottlenecks. While this dissertation resolves most fundamental issues associated with composable architectures, a few more optimizations are still possible, which will also be discussed briefly.

11.1 Dissertation Contributions

This dissertation studies and addresses inefficiencies and bottlenecks in previous EDGE architectures. Early EDGE designs, TRIPS and TFlex, have a number of serious performance bottlenecks [32]. They distribute the instructions in each in-flight dataflow block among all participating cores, increasing cross-core operand communication latency and consumed energy. These architectures exploit pipelines of large blocks, and thus next-block misprediction flushes were particularly expensive in terms of delay and energy consumed for refilling the blocks after each misprediction. Additionally, because EDGE blocks are multi exit, next-block predictor had low accuracy compared to conventional taken/not taken predictors. The intra-block branches that are converted to predicates, are evaluated in the execution stage, but would have been predicted as branches in a conventional processor. Finally, for high fanout operands, the early EDGE compiler generates trees of *move* instructions in oder to handle operand fanout using dataflow. These overhead move instructions consume additional power and increases the dependence height in the program's critical path.

By addressing these issues, the T3 dynamic multicore EDGE architecture proposed in this dissertation operates efficiently in a wide spectrum of energy and performance operating points ranging from low-power to high-performance. To achieve a high degree of energy efficiency, T3 addresses several fundamental issues associated with composable block-based dataflow execution:

- By mapping each dataflow block to only one executing core [67], T3 halves the cross-core network traffic by eliminating cross-core dataflow communication, which is a major bottleneck in previous EDGE architectures. As a result, intra-block dataflow operand delivery is only used within cores and only inter-block register communication occurs between cores.

- The Iterative Path Predictor [69] solves the low multi-exit next bock prediction accuracy and low speculation rate problems caused by heavily predicated execution. By predicting multiple branches in the same block very quickly, this predictor predicts the taken predicate path within each block and then uses that estimated path to predict the predicates instead of evaluating them in the execution stage. Additionally, using this predicted predicate path improves the next block prediction accuracy compared to TFlex.

- The Exposed Operand Broadcasts [69] proposed by Li et al. [48, 69] address another major issue, the energy consumed and latency incurred by compiler-generated trees of move instructions built for wide-fanout operands. Exploiting both low-overhead architecturally exposed broadcasts and direct dataflow communication, T3 supports fast and energy-efficient operand delivery for high- and low-fanout instructions. By exposing different operand delivery mechanisms to the compiler, T3 reduces the total energy consumed for operand delivery beyond what both dataflow machines and hybrid operand delivery in superscalar machines can achieve.

- T3 employs register bypassing [68] to speedup cross-core inter-block register communication by bypassing late register outputs between producer and consumer blocks. Relying on block-based EDGE execution model, different from previous cross-core register bypassing mechanism used by other composable designs [46], this method does not require a synchronizing score board nor shared buses.

- By reissuing previously executed or flushed blocks while they are still in the instruction queue [68], T3 saves both delay and energy by skipping fetch and decode of the reissued blocks. As T3 relies on statically formed dataflow blocks, this mechanism does not incur any major hardware overhead for finding global re-convergent points as done by trace caches dynamically [71].

Exploiting these novel mechanisms, T3 demonstrates significant performance and energy advantages over previous composable EDGE designs. Furthermore, T3 achieves high energy efficiency at different power and performance operating points across a wide power/performance spectrum. T3 extends the power/performance tradeoffs beyond what conventional processors can offer using traditional voltage and frequency scaling. These features make T3 an attractive candidate for a wide range of control- and memory-intensive workloads under varying power and performance constraints.

In order to design and guide these new optimization mechanisms, this dissertation proposes a methodical approach [68] using critical path analysis to detect performance and scalability bottlenecks of these designs. The criticality analysis indicates that using these optimizations, T3 almost eliminates most previously detected bottlenecks such as cross-core communication, fetch stalls, delayed predicates and fanout moves. As the only major T3 critical resource is instruction execution, further improvements are still possible.

11.2 Future Directions

While this dissertation has contributed a good deal towards designing energy-efficient dynamic multicore EDGE architectures, there is more research to be done. Additional research work could improve the compilers code for this new architecture. Also, the ISA and microarchitecture can be further optimized for energy efficiency. We outline these areas of future work here.

11.2.1 Compiler Improvements

This dissertation only investigates methods to improve EDGE microarchitecture and ISA for better power efficiency. Evaluating the sensitivity of the proposed mechanisms to different compiler analyses and optimizations is one interesting future direction. For example, different block formation strategies [50] can directly affect the accuracy and performance of our iterative path predictor proposed in Chapter 8. Also, as EDGE architectures depend partly on compiler technology to obtain performance and power efficiency from a range of microarchitecture features, we believe our results can be further improved by employing more fitting and advanced compiler techniques. This section discusses two of the possible opportunities.

11.2.1.1 Improved Instruction Placement

Instruction scheduling [20] used by the EDGE compiler [78] employs Spatial Path Scheduling (SPS) algorithm for assigning instruction IDs to instructions in each block. In each pass, this algorithm assigns a *placement cost* to each instruction and uses these assigned costs to prioritize the most critical instructions and find the location and criticality for each instruction in the block. When using deep mapping, these IDs determine the location of the instructions in the issue queue (reservation stations) of the core executing that block, and so can affect the order, in which instructions are issued. Given that each T3 is 2-wide issue, if more than two instructions are ready in a given cycle, the instruction IDs determine which instruction must be issued first.

In evaluating this cost value for each instruction, the SPS algorithm also considers the costs assigned to the instructions on which the current instruction is dependent. When enabling predicate prediction in T3, the predicate test instructions are predicted, and so the data dependences between them and their producing instructions, register reads or memory accesses, will not exist any more. An SPS algorithm revised for T3 should consider this observation when assigning the placement cost to all predicate tests instruction and also its producing instructions. For example, when IPP is enabled, if a predicted

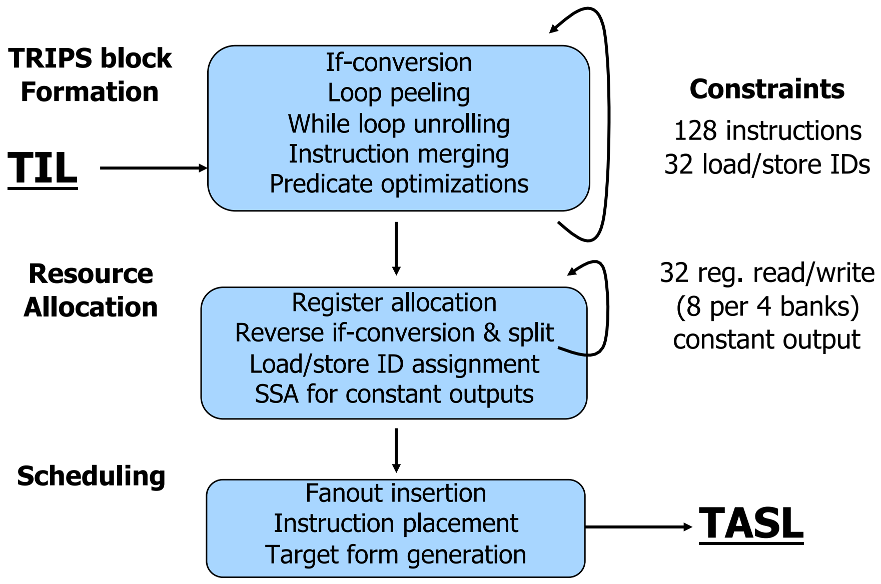


Figure 11.1: TRIPS compiler overview.

predicate is on the critical path, its producing register read is less likely to also be on the critical path because the predicate is predicted and does not wait for that register value. Consequently, the register read should be given less priority by the scheduling algorithm.

11.2.1.2 Better Code Generation

We anticipate that our results can further improve by employing a highly tuned production compiler rather than our current research compiler [78]. As shown in Figure 11.1, the current TRIPS compiler back end [78] has three major phases. The first phase is block formation [50], in which the compiler combines basic blocks into a set of EDGE blocks. This phase uses if-conversion, predication, unrolling, tail duplication, and head duplication as necessary to form optimized blocks [52]. The compiler also performs scalar optimizations that merge redundant instructions and eliminate unnecessary predicates as an integrated step of block formation. During block formation, the compiler assumes an infinite virtual register set and uses a RISC-like intermediate form called TIL (TRIPS Intermediate Language). The second phase applies ISA and microarchitectural restrictions such as register allocation and load/store ID assignments. The last phase in the TRIPS compiler backend is instruction scheduling, which outputs TRIPS Assembly Language (TASL) after assigning instruction IDs of instructions in each block.

This research compiler can be improved in several aspects. First, the compiler does not consider any high-level front end related information when performing block formation for EDGE. Second, the block formation, register allocation and instruction scheduling happen in completely different phases whereas they are indeed inter related. An alternative solution is to use a production front end and back end and then translate the fully optimized code into EDGE ISA. To determine how much additional performance and energy efficiency is achievable with better compilation, a highly optimized compiler is under developement.

11.2.2 E2: Next EDGE ISA and Microarchitecture

In addition to improving code generation, other optimizations are possible via ISA and microarchitectural extensions. The next EDGE ISA and microarchitecture, which is called E2 [59], incorporates variable-size blocks [50] and support for SIMD and vector operations in addition to the EOBs, IPP and other T3 optimizations. Figure 11.2 shows the basic architecture of an E2 microarchitecture with 32 cores, and a block diagram of the internal structure of each E2 core. An E2 core contains four lanes, with each lane consisting of a 64-bit ALU and one bank of the instruction window (with the capacity needed for 32 instructions), operand buffers, and register file. ALUs support both integer and floating point operations, as well as fine-grained SIMD execution (eight 8-bit, four 16-bit, or two 32-bit integer operations per cycle, or two single-precision floating point calculations per cycle). Breaking the window into these four lanes allows high vector throughput with little additional hardware complexity. It also facilitates the support for variable-size blocks. This section briefly discusses these novel features in E2.

11.2.2.1 Variable Block Sizes

Variable-size blocks [50] improve resource utilization and power efficiency in E2. The block size is a multiple of the size of each instruction lane. So, this ISA supports four possible block sizes of 32, 64, 96 and 128 instructions. The total capacity of the instruction queue of each E2 core (4 lanes together) is 128 instructions, which is the maximum block size. Each E2 core can allocate and run up to 4 blocks at the same time (four 32-instruction blocks). Consequently, an E2 architecture with 16 composed cores can support 64 in-flight blocks. This high number of in-flight blocks and flexibility of each E2 core for mapping multiple blocks result in a large space of possi-

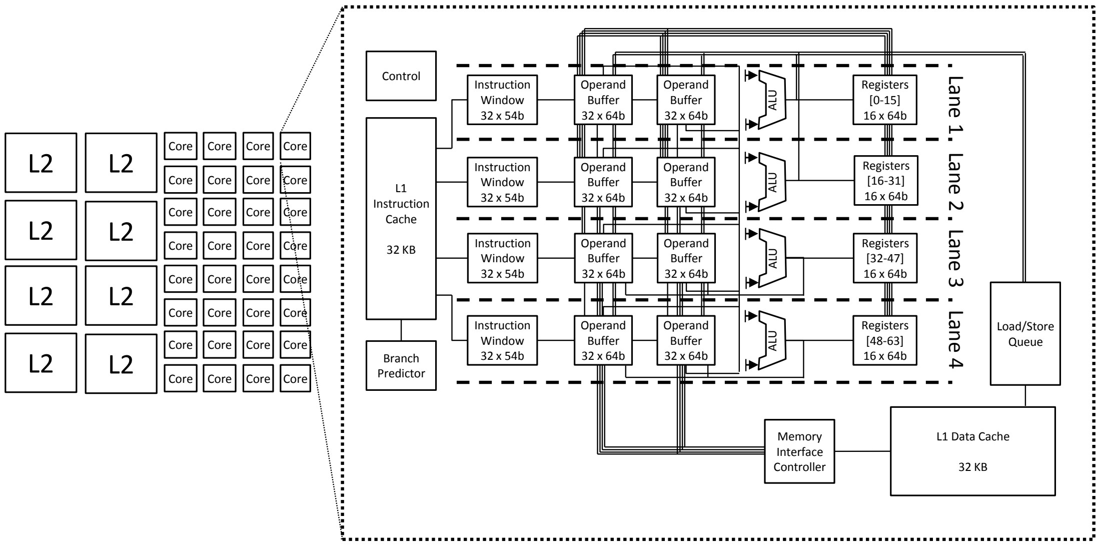

L2
Core
Control
Instruction Window 32 x 54b
Operand Buffer 32 x 64b
ALU
Registers [0-15] 16 x 64b
Registers [16-31] 16 x 64b
Registers [32-47] 16 x 64b
Registers [48-63] 16 x 64b
Lane 1
Lane 2
Lane 3
Lane 4
L1 Instruction Cache 32 KB
Branch Predictor
Load/Store Queue
Memory Interface Controller
L1 Data Cache 32 KB

Figure 11.2: E2 microarchitecture block diagram[59].

ble policies for distributed block mapping and next-block prediction protocols. An effective policy should be able to sustain a high prediction accuracy, low instruction cache miss rate and high number of in-flight blocks. Exploring different policies and mechanisms for block mapping and next-block prediction for such a high number of in-flight blocks is an interesting future research direction.

11.2.2.2 SIMD/Vector Optimizations

E2 also supports instruction set extensions such as SIMD/vector operations [59] and out-of-order execution of both vectors and scalars. The E2 instruction set and execution model supports three new capabilities that enable efficient vectorization across a broad range of codes. First, the E2 compiler slices up the issue window into vector lanes and so it can achieve highly concurrent, out-of-order issue of mixed scalar and vector operations with lower energy overhead than scalar mode. Second, the statically allocated reservation stations treat the issue window as a vector register file, with wide fetches to memory and limited copying between a vector load and the vector operations. Third, similar to T3, the atomic block-based execution in E2 allows reissuing of both vector and scalar instruction blocks mapped to different lanes of the issue window. This enables repeated vector operations to issue with no fetch or decode energy overhead after the first loop iteration [59].

E2 cores operate in the scalar mode or vector mode. In the scalar mode, an instruction can send operands to any other instruction in the block, and two of the four ALUs are turned off to save power. In the vector mode, all four ALUs are turned on, but instructions can only send operands to instructions in the same lane. The mode is determined based on a bit in each block header. So two consecutive blocks could configured in two separate modes. Moreover, since each block is mapped to one core, each core can adapt quickly to different application phases on a block-by-block basis. We expect that the combination of better compilation and common-practice ISA extensions will further enhance the capabilities of EDGE architectures.

Bibliography

[1] Low power optimization datasheet for intel Atom processor, Embedded Insights, www.embeddedinsights.com/ei_download.php?file=basicslpdint01.

[2] The standard performance evaluation corporation (SPEC), http://www.spec.org/.

[3] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Fermi. http://nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_ Compute_Architecture_Whitepaper.pdf, 2009.

[4] AMD. HD 6900 Series Instruction Set Architecture. http://developer.amd.com/ gpu/amdappsdk/assets/AMD_HD_6900_Series_Instruction_Set_Architecture.pdf, February 2011.

[5] Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, and Haitham H. Akkary. Transparent control independence (tci). In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 448–459, San Diego, California, USA, 2007.

[6] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Computer*, 39(3):300–318, March 1990.

[7] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th*

Symposium on Microarchitecture, pages 92–103, Research Triangle Park, North Carolina, United States, December 1997.

[8] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *International Symposium on Computer Architecture*, pages 275–286, San Diego, California, June 2003.

[9] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 237–248, Austin, Texas, December 2001.

[10] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *33rd International Symposiumon Microarchitecture*, pages 337–347, Monterey, California, December 2000.

[11] Eric Borch, Srilatha Manne, Joel Emer, and Eric Tune. Loose loops sink chips. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 299, Washington, DC, USA, February 2002.

[12] Robert S. Boyer and J Strother Moore. MJRTY - a fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe, of Automated Reasoning Series*, pages 529–543, 1977.

[13] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Computer Architecture News*, 28(2):83–94, June 2000.

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

[15] R. Canal, J. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. In *6th International Symposium on High Performance Computer Architecture*, pages 133–142, Toulouse, France, January 2000.

[16] Ramon Canal and Antonio González. A low-complexity issue logic. In *Conference on Supercomputing*, pages 327–335, Santa Fe, New Mexico, May 2000.

[17] Ramon Canal and Antonio González. Reducing the complexity of the issue logic. In *Conference on Supercomputing*, pages 312–320, Sorento; Italy, June 2001.

[18] Chen-Yong Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 4–15, Austin, Texas, December 2001.

[19] Weihaw Chuang and Brad Calder. Predicate prediction for efficient out-of-order execution. In *the 17th Annual International Conference on Supercomputing*, pages 183–192, San Francisico, June 2003.

[20] Katherine E. Coons, Xia Chen, Doug Burger, Kathryn S. McKinley, and Sundeep K. Kushwaha. A spatial path scheduling algorithm for EDGE architectures. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 129–140, San Jose, December 2006.

[21] Katherine E. Coons, Behnam Robatmili, Matthew E. Taylor, Betrand A. Maher, Doug Burger, and Kathryn S. McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *The 17th Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 32–42, Toronto, Canada, October 2008.

[22] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. *SIGARCH Comput. Archit. News*, 28(2):316–325, June 2000.

[23] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer architecture*, pages 126–132, New York, December 1975.

[24] Jeff Diamond, Behnam Robatmili, Stephen W. Keckler, Kazushige Goto, Doug Burger, and Robert van de Geijn. High performance dense linear

algebra on spatially partitioned processors. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 63–72, Salt Lake City, February 2008.

[25] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen Blackburn, and Kathryn McKinley. Looking back on the language and hardware revolution: Measured power performance, and scaling. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–332, Newport Beach, California, 2011.

[26] E.Oezer, S. Banerjia, and T. Conte. A new approach to scheduling for clustered registerfile microarchitectures. In *The 31st International Symposiumon Microarchitecture*, pages 308–315, Dallas, December 1998.

[27] K. L. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing processor cycle time through partitioning. In *30th International Symposium on Microarchitecture*, pages 327–356, North Carolina, December 1997.

[28] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture*, pages 74–85, July 2001.

[29] Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *International Symposium on Microarchitecture*, pages 228–240, San Diego, December 2003.

[30] Brian R. Fisk and R. Iris Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *International Conference on Computer Design*, pages 538–545, October 1999.

[31] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, Portland, OR, December 1992.

[32] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul V. Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the TRIPS computer system. In *The 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–12, March 2009.

[33] Dan Gibson and David A. Wood. Forwardflow: a scalable core for power-constrained CMPs. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 14–25, Saint-Malo, France, June 2010.

[34] Madhu Saravana Sibi Govindan. *E3 : Energy-Efcient EDGE Architectures*. PhD thesis, Austin, TX, USA, 2010. Supervised by Stephen W. Keckler.

[35] Madhu Saravana Sibi Govindan, Behnam Robatmili, Hadi Esmaeilzadeh, Bertrand Maher, Dong Li, Aaron Smith, Stephen W. Keckler, and Doug Burger. Scaling power and performance via processor composability. Technical report, 2010. UT Austin, Department of Computer Sciences TR-10-14.

[36] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. In *IEEE Computer*, volume 41, pages 33–38, July 2008.

[37] Andrew D. Hilton and Amir Roth. Ginger: Control independence using tag rewriting. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 436–447, San Diego, June 2007.

[38] Michael Huang, Jose Renau, and Josep Torrellas. Energy-efficient hybrid wakeup logic. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 196–201, Monterey, California, August 2002.

[39] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, San Diego, June 2007.

[40] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *3rd IEEE Sym-*

posium on High-Performance Computer Architecture, HPCA, pages 218–229, San Antonio, February 1997.

[41] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture*, pages 381–394, Chicago, December 2007.

[42] Ho-Seop Kim and James E. Smith. Instruction level distributed processing. In *IEEE Computer 34, 4*, pages 59–65, 2001.

[43] Ho-Seop Kim and James E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *International Symposium on Computer Architecture*, pages 71–81, Anchorage, Alaska, May 2002.

[44] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc Processor. *IEEE Micro*, 25(2):21 – 29, 2005.

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

[46] Venkata Krishnan and Josep Torrellas. Fast communication in hardware-based speculative chip multiprocessors. *International Journal of Parallel Programing*, 29(1):3–33, February 2001.

[47] 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 *International Symposium on Microarchitecture*, pages 81–93, December 2003.

[48] Dong Li, Behnam Robatmili, and Doug Burger. Hybrid operand communication for dataflow processors. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures, in conjunction with 35th Intl. Symposium on Computer Architecture*, pages 61–71, Austin, TX, June 2009.

[49] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, New York, New York, December 2009.

[50] Bert Maher. *Atomic Block Formation for Explicit Data Graph Execution Architectures.* PhD thesis, Austin, TX, USA, 2010. Supervised by Doug Burger.

[51] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *22nd annual international symposium on Computer architecture*, pages 138–150, Santa Margherita Ligure, Italy, May 1995.

[52] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Microarchitecture*, pages 45–54, November 1992.

[53] S Melvin and Yale Patt. Enhancing Instruction Scheduling With a Block-Structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, June 1995.

[54] S. W. Melvin, M. C. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *Workshop on Microprogramming and Microarchitecture*, pages 60–63, Austin, TX, November 1988.

[55] Andreas Moshovos and Gurindar S. Sohi. Speculative memory cloaking and bypassing. *Int. J. Parallel Program.*, 27(6):427–456, 1999.

[56] R. Nagarajan, Xia Chen, R.G. McDonald, D. Burger, and S.W. Keckler. Critical path analysis of the TRIPS architecture. In *International Symposium on Performance Analysis of Systems and Software*, pages 37–47, March 2006.

[57] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25(2):206–218, May 1997.

[58] Dmitry V. Ponomarev, Gurhan Kucuk, Oguz Ergin, Kanad Ghose, and Peter M. Kogge. Energy-efficient issue queue design. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(5):789–800, October 2003.

[59] Andrew Putnam, Aaron Smith, and Doug Burger. Dynamic vectorization in the e2 dynamic multicore architecture. *SIGARCH Comput. Archit. News*, 38:27–32, January 2011.

[60] Eduardo Quinones, Joan-Manuel Parcerisa, and Antonio Gonzalez. Selective predicate prediction for out-of-order processors. In *Conference on Supercomputing*, pages 46–54, Tampa, June 2006.

[61] Eduardo Quinones, Joan-Manuel Parcerisa, and Antonio Gonzalez. Improving branch prediction and predicated execution in out-of-order processors. In *International Symposium on High Performance Computer Architecture, 2007*, pages 75–84, Phoenix, Arizona, February 2007.

[62] Marco A. Ramirez, Adrian Cristal, Mateo Valero, Alexander V. Veidenbaum, and Luis Villa. A new pointer-based instruction queue design and its power-performance evaluation. In *International Conference on Computer Design*, pages 647–653, Las Vegas, Nevada, October 2005.

[63] Marco A. Ramírez, Adrian Cristal, Alexander V. Veidenbaum, Luis Villa, and Mateo Valero. Direct instruction wakeup for out-of-order processors. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 2–9, January 2004.

[64] Nitya Ranganathan, Doug Burger, and Stephen W. Keckler. Analysis of the TRIPS Prototype Block Predictor. In *International Symposium on Performance Analysis of Systems and Software*, pages 195 – 206, April 2009.

[65] Behnam Robatmili, Katherine Coons, and Doug Burger. Balancing local and global parallelism for single-thread applications in a composable multi-core system. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures, in conjunction with 34th Intl. Symposium on Computer Architecture*, pages 2–10, Beijing, China, June 2008.

[66] Behnam Robatmili, Katherine E. Coons, Doug Burger, and Kathryn S. McKinley. Register bank assignment for spatially partitioned processors. In *21st Annual Languages and Compilers for Parallel Computing Workshop (LCPC)*, pages 64–79, Edmonton, Alberta, Canada, July 2008.

[67] Behnam Robatmili, Katherine E. Coons, Doug Burger, and Kathryn S. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *International Symposium on Microarchitecture (MICRO)*, pages 23–34, November 2008.

[68] Behnam Robatmili, Madhu Saravana Sibi Govindan, Doug Burger, and Steve Keckler. Exploiting criticality to reduce bottlenecks in distributed uniprocessors. In *17th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 431–442, San Antonio, December 2011.

[69] Behnam Robatmili, Dong Li, , Hadi Esmaeilzadeh, Madhu Saravana Sibi Govindan, Doug Burger, and Steve Keckler. T3: An Energy-Efcient Dynamic Multicore Architecture. submitted to The 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44), 2011.

[70] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35, Paris, France, 1996.

[71] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *International Symposium on Microarchitecture*, pages 138–148, December 1997.

[72] Pierre Salverda and Craig Zilles. A criticality analysis of clustering in superscalar processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, Barcelona, Spain, December 2005.

[73] Pierre Salverda and Craig Zilles. Fundamental performance challenges in horizontal fusion of in-order cores. In *International Symposium on High-Performance Computer Architecture*, pages 252–263, Salt Lake City, UT, 2008.

[74] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, and Doug Burger. Universal mechanisms for data-parallel architectures. In

International Symposium on Microarchitecture, pages 303–314, December 2003.

[75] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture*, pages 422–433, San Diego, June 2003.

[76] Andre Seznec. The O-GEHL branch predictor. In *Journal of Instruction-Level Parallelism (JILP) Special Issue: The first JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[77] Andre Seznec. A 256 kbits L-TAGE branch predictor. In *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 2007.

[78] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, Newyork, March 2006.

[79] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. *SIGARCH Comput. Archit. News*, 25(2):194–205, 1997.

[80] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 521–532, Santa Margherita Ligure, Italy, June 1995.

[81] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *International Symposium on Computer Architecture*, pages 132–144, June 2001.

[82] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *International Symposium on Microarchitecture*, pages 148–159, November 1998.

[83] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, 2002.

[84] Suriya Subramanian and Kathryn S. McKinley. HeDGE: hybrid dataflow graph execution in the issue logic. In *The 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 308–323, Cyprus, January 2009.

[85] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. WaveScalar. In *36th Symposium on Microarchitecture*, pages 291–302, San Diego, CA, December 2003.

[86] D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-order execution using simple in-order cores. Technical Report CS-2007-11, University of Virginia, Department of Computer Science, August 2007.

[87] D. Tarjan, S. Thoziyoor, and N. Jouppi. HPL-2006-86, HP Laboratories, Technical Report. 2006.

[88] Matthew E. Taylor, Katherine E. Coons, Behnam Robatmili, Doug Burger, and Kathryn S. McKinley. Policy search optimization for spatial path planning. In *Workshop on Machine Learning for Systems Problems (NIPS)*, Vancouver, Canada, December 2007.

[89] Matthew E. Taylor, Katherine E. Coons, Behnam Robatmili, Bertrand A. Maher, Doug Burger, , and Kathryn S. McKinley. Evolving compiler heuristics to manage communication and contention. In *Twenty-Fourth Conference on Artificial Intelligence (Nectar Track)*, pages 1690–1693, Atlanta, GA, July 2010.

[90] E. Tune, Dongning Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *International Symposium on High Performance Computer Architecture*, pages 181–195, Monterrey, Mexico, January 2001.

[91] E. Tune, D. M. Tullsen, and B. Calder. Quantifying instruction criticality. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, Charlottesville, Virginia, September 2002.

[92] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Bring it all to software: Raw machines. *IEEE Computer*, 30(9), 1997.

[93] Yasuko Watanabe, John D. Davis, and David A. Wood. WiDGET: Wisconsin decoupled grid execution tiles. In *the 37th annual international symposium on Computer architecture*, pages 2–13, Saint-Malo, France, June 2010.

[94] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *IEEE 13th International Conference on High Performance Computer Architecture*, pages 25–36, Phoenix, Arizona, February 2007.

[95] V. Zyuban and P. Kogge. Optimization of high-performance superscalar architectures for energy efficiency. In *2000 International Symposium on Low Power Electronics and Design*, pages 84–89, Rapallo, Italy, July 2000.

Vita

Behnam Robatmili was born in Arak, Iran on 28 July 1978. He received the Bachelor and Master of Science degrees in Computer Engineering from the University of Tehran in 2001 and 2004, respectively. He entered the Ph.D. program at the University of Texas at Austin in the August 2005.

Permanent address: 3477 Lake Austin Blvd, Apt A
Austin, Texas 78758

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.