Efficient Throughput Cores for Asymmetric Manycore Processors

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Engineering

by

David Tarjan

August 2009

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Engineering

David Tarjan

Approved:

Kevin Skadron (Advisor)

Sudhanva Gurumurthi	Mircea R. Stan (Chair)
John C. Lach	Lloyd R. Harriot

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

August 2009

Abstract

The microprocessor industry has had to switch from developing ever more complex and more deeply pipelined single-core processors to multicore processors due to running into power, thermal and complexity limits.

Future microprocessors will be asymmetric manycore chip multiprocessors, with a small number of complex cores for serial programs and serial sections of parallel programs. The majority of the cores will be small, power- and area-efficient cores to maximize overall throughput in a limited power budget.

The main contributions of this dissertation are techniques for improving the performance and area-efficiency of these throughput-oriented cores. This work shows how the single-thread performance of small, scalar cores can be increased or dynamically combined to speed up programs with only a limited number of parallel threads. It also shows how to improve both the cores and the cache subsystem of multicore processor using SIMD cores.

Acknowledgments

I would like to thank my advisor Kevin Skadron for his patience, advice and encouragement throughout my graduate career. Kevin kept me on track and gave me the impetus to move forward when I was stuck in a rut.

I also want to express my gratitude to the members of the Lava Lab group, Michael Boyer, Shuai Che, Jiayuan Meng, Lukasz Szafaryn and Jeremy Sheaffer for their work on all of our joint projects and for generally putting up with all of my questions on a myriad of subjects. I would like to thank Sudhanva Gurumurthi, Mircea Stan, Lloyd Harriott and John Lach for serving on my committee and for on enlightening me about their areas of expertise.

Finally, I have to thank my parents for their continuing support and understanding.

The research in this dissertation was made possible by support from the National Science Foundation under grant numbers CCR-0306404, CNS-0509245, CCR-0133634 (CAREER), CCF-0429765, EIA-0224434 and IIS-0612049; from the Semiconductor Research Corporation under grant 1607.001, an NVIDIA research grant and a research grant from Intel MTL. I was also supported by a William A. Ballard Fellowship and an Award for Excellence in Scholarship in the Sciences and Engineering from the University of Virginia.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Until recently the main goal for processor designers was to improve single-thread latency with each successive processor generation. Each processor contained a single-core, and architects took advantage of Moore's Law [76] to increase both the sophistication and the frequency of this core with each generation. By improving the sophistication of the core, the core was capable of executing more instructions per cycle (through extracting higher levels of instruction-level-parallelism (ILP)) and the higher frequency allowed the core to execute more cycles per second. These two trends together allowed mainstream processors to execute instructions at an ever higher rate. Between 1986 and 2002 single-thread performance improved at a 52% annual rate [9].

The great benefit of this trend was that software written for old processors could always take advantage of the higher performance offered by new processors without having to change any of their core algorithms.

The cost of this trend was that both the complexity of processors and their power consumption grew rapidly along with their performance. The sheer size of these complex processors posed limitations to their further success, as it was getting harder and harder for different parts of a processor to communicate with each other inside a single cycle [37].

In addition, the techniques used to extract higher per-cycle performance were running into diminishing returns [2], with small performance increases requiring very large increases in core area. But the real limiting factor turned out to be a combination of the growing complexity of processors and fundamental changes in how the underlying CMOS technology was scaling.

1.1 CMOS Scaling and its Implications for Processor Architecture

Even though the term Moore's Law [76] is used colloquially to refer to all the improvements in semiconductor technology over time, a much more prescriptive explanation of the continuing improvements in CMOS technology, which has been the mainstream technology for semiconductor devices for approximately 30 years now, is *Dennard scaling*.

Dennard scaling [33] describes scaling rules for CMOS transistors which improve density, frequency and power at the same time. These scaling rules (shown in Table 1.1) require that all the important physical dimensions of a transistor are scaled by the same factor (κ). To maintain a constant electrical field strength and a constant power density as the size of the transistors is scaled down, the supply voltage V_{dd} is also scaled down by κ. The threshold voltage V_{th} also needs to be scaled down by κ to have the current I scale down by κ [92].

Since the advent of CMOS technology, the semiconductor industry has roughly succeeded in following Dennard scaling. As a consequence, microprocessors built in CMOS technology enjoyed ever higher frequencies and transistor counts, increasing the performance of microprocessors over time independently of improvements in the design of microprocessors. The power consumption and power density of microprocessors did not stay

Variable	**Scaling Factor**
Oxide Thickness t_{ox}, Gate Width and Length W & L	$1/\kappa$
Doping Concentration N_a	κ
Supply and threshold voltage V_{dd} & V_{th}	$1/\kappa$
Current I	$1/\kappa$
Capacitance $\varepsilon A/t_{ox}$	$1/\kappa$
Delay time/circuit VC/I	$1/\kappa$
Power dissipation/circuit VI	$1/\kappa^2$
Power density VC/A	1

Table 1.1: Dennard scaling rules for the main variables affecting MOSFET transistors.

constant, as would be expected from simple Dennard scaling, because designers opted for larger dies and more aggressive pipelining (leading to higher frequencies) [52] to achieve even higher performance levels.

This trend was greatly worsened in recent years as the reduction in supply voltage has slowed down significantly compared to the continued scaling in feature size. With supply voltage scaling more slowly than transistor density, power density and total power dissipation became the primary limiting factors for microprocessors.

The primary reason that supply voltage scaling has slowed down is that a certain ratio of supply voltage to threshold voltage needs to be maintained to run a transistor at optimal speed. The problem is that scaling of the threshold voltage V_{th} has almost stopped. The scaling of threshold voltage in turn is primarily limited by subthreshold leakage, that is the amount of current flowing through a CMOS transistor even when it is turned off. Since subthreshold leakage is exponentially dependent on the subthreshold voltage,

$$I_{sub} \propto e^{-V_{th}} \tag{1.1}$$

as shown in equation 1.1, there comes a point where further lowering of the threshold voltage leads to unacceptable increases in subthreshold leakage. For a given generation of CMOS technology the amount of subthreshold leakage can be modified in several ways.

The most obvious is to produce transistors with higher threshold voltages, but these have slower switching speeds than those with regular threshold voltages. There are many other possibilities, but all of them either increase the cost of production or negatively effect the speed of the transistors. Overall, no currently available technology can decrease leakage fast enough for the normal scaling of threshold voltage to continue at anywhere close to its historical rate.

Designers responded these trends by designing the current generation of microprocessors to be more power efficient. They did this in several ways:

- They backed off the very aggressive pipelining that had been used in the last generation of microprocessors (such as the Pentium 4 [50, 14]) to decrease power-density.
- They slowed down frequency scaling drastically for even these less aggressively pipelined cores, such that today only a single commercial product (the IBM Power 6 [109]) exceeds 4 GHz in clock speed, even though a 3.8 GHz Pentium 4 [14] was introduced in 2004. Thanks to much less aggressive frequency targets, it was possible to be much more aggressive in replacing regular threshold and low threshold transistors with high threshold transistors throughout large parts of the core logic of many recent processors.
- The size of new core designs was either scaled back or old design were just shrunk with technology. For example, two Intel Core 2 Duo [91] cores, which is the successor to the Pentium 4, occupy a similar area as a single Pentium 4 Prescott [14] when compared at the same technology node. core)
- Single-thread performance was de-emphasized in several designs [3, 56], which meant that the size and complexity of the cores could be scaled back to the level of the early 1990's.

1.2 Motivation for Chip Multi-Processors (CMPs)

The pull-back from ever larger and ever faster single cores left the designers with the question of how to best use the continuing increase in transistor numbers. One option was to just dedicate an ever larger portion of the die area to caches, but since caches have diminishing returns with increasing size, the end user would see only a very small increase in performance. Another option was to integrate more system functionality on the CPU die itself. This strategy is being followed by many companies, but the number of system components which show a big benefit from having a very high-speed, low latency connection to the CPU core is limited.

The option which the industry as a whole adopted was to integrate multiple cores on a single die, hoping that there were enough parallel programs which could take advantage of multiple cores per chip, even if single-thread latency was not decreased. The name chip multi-processor (CMP) has become popular for such designs.

The first and second generation of CMPs has used core designs from the last generation of single-core processors or new designs of similar complexity. This limited the number of cores per die to two initially, and four cores with the next process generation, while single-thread performance stayed constant or increased marginally over the last generation of single-core processors.

1.3 Why Asymmetric Manycore CMPs?

With the transition from single-core processors to CMPs, programs have to be parallel to take advantage of the increasing throughput offered by CMPs. As the number of cores per die increases over time and the performance of a single core increases only slowly or even stagnates, the gap between serial and parallel programs will grow wider and wider.

But even if all programs are parallelized, the maximum number of threads that each program will be able to take advantage of at any given point in time will vary widely. Another problem is that even programs which can take advantage of a virtually unlimited number of threads will have some portion of their execution time be serial. As Hill and Marty point out [49], the overall speedup for such programs is quickly limited by the serial portion of the program.

Computer architects are faced with a number of questions when deciding the high-level design of future processors.

How many cores should be on the die, and what is their individual single-thread performance? There is a clear tradeoff between more and less performant cores, as many structures of high performance cores increase super-linearly in complexity with increasing performance [75]. This relationship holds true not just for area, but also for power-efficiency, as higher performance cores are typically also less power efficient [46]. For applications with sufficient parallelism, Davis *et al.* [32] and Carmean [25] show that maximum aggregate throughput is achieved by using a large number of highly multithreaded scalar cores. Designs which sacrifice single-thread performance and have a large number of small, simple cores have been called *manycore* processors, to distinguish such designs from the current multicore designs, which have fewer, larger cores.

How far should changes in the cores go to maximize power- and area-efficiency? Since the workload of a manycore CMP consists of parallel programs, the question becomes if a different, more parallel ISA could be used to make further gains in efficiency. Single Instruction Multiple Data (SIMD) organization are such an option. A SIMD core the same instruction for multiple threads in lockstep. See Section 1.4 for a more detailed discussion of SIMD architectures.

Once a program has been expressed in a parallel fashion and can execute using multiple threads, the changes to execute the same program on SIMD are relatively minor. This is especially true for array type SIMD ISAs since they present the programmer with the abstraction of a number of normal, sequential programs, which just execute faster if all threads follow the same control-flow path. SIMD ISAs are an attractive choice compared to scalar ISAs for the small, throughput-oriented cores , since cores implementing these ISAs can amortize both the area and power of a core's frontend over multiple backends. The power and area benefits of SIMD ISAs have been shown in shown in recent graphics processing units(GPUs) [68, 74], the Imagine [60] and Merrimac [31] architectures, and the Cell processor's [51] Synergistic Processing Elements [38]. Note that SIMD cores are not well suited for programs where each thread has very different control-flow, as this type of program will underutilize the SIMD hardware. The recently announced Intel Larrabee architecture [97] tries to deal with this problem by combining wide SIMD units with small scalar cores, allowing such a processor to have high performance on a wider variety of workloads.

Should there be only a single type of core for the whole chip, or a mix of large and small cores? Work by Hill and Marty [49], Kumar *et al.* [67] and Suleman *et al.* [112] shows that having at least a single high performance core is very beneficial for overall throughput, even if the high performance core uses the area and power of multiple smaller cores. CMPs with a mix of large and small cores are called *asymmetric* or *heterogeneous* CMPs.

My extrapolation from the above mentioned body of work is that a future CMP will have a combination of core types. It will have a small number of large, high-performance cores and a large number of small, throughput cores. The throughput cores will themselves be divided between scalar and SIMD cores, to provide a maximum of flexibility and power- and area-efficiency.

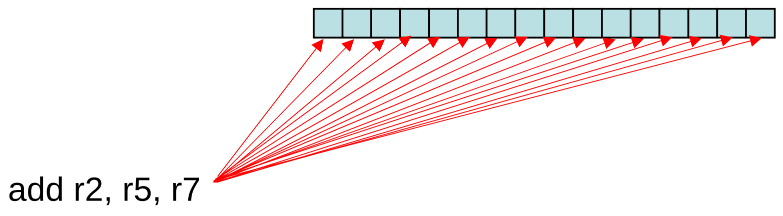

Figure 1.1: A SIMD core broadcasts a single instruction to many threads, which execute the instruction in lockstep.

1.4 A Short Primer on SIMD

As shown in Figure 1.1 SIMD architectures execute a single instruction on multiple pieces of data, but the way this capability is expressed to the programmer divides the architectures into two classes: vector and array architectures.

Vector computers expose to the programmer vector registers, which hold multiple data words. Instructions operate on these large registers, accomplishing more work per instruction than to scalar machines. Each scalar word in a vector register has a specific position in the vector, and this position along with the associated execution resources are often called vector lanes. The programmer has to specify explicitly how to load values into these large registers, either from a contiguous chunk of memory or using arbitrary per vector lane addresses. The later operation is often called a gather (if the operation is a load) or a scatter (if the operation is a store). If there are branches which depend on per vector lane values, the programmer has to manually make sure that only the vector lanes taking a particular branch receive updated values. This is often accomplished using a bitmask called the active mask.

Array computers in the tradition of the Illiac IV [15] present the abstraction that each core is made up of a number of separate processing elements (PE), each of which is scalar. Instructions are fetched through a separate unit and broadcast to each individual PE. If a PE does not follow a conditional branch it simply does not execute instructions which are broadcast while the rest of the PEs execute the branch. Similarly, scatter/gather operations are simply normal stores and loads.

The advantage of vector processors is that direct register to register communication in different lanes is expressed more naturally (so called vector permutations), while array processors are more natural in dealing with divergent control flow and scatter/gather operations.

SIMD architectures have a built in advantage in how they scale performance with Moore's Law. In contrast to scalar architectures, which require techniques like increased pipelining, larger caches or sophisticated ILP extraction techniques to increase performance, SIMD architectures can in theory increase performance linearly simply by increasing their SIMD width. This means that the area-efficiency of SIMD architectures stays the same or even increases as they increase their performance. Area-efficiency increases as the constant area of a single frontend is amortized over a larger number of backends. I say in theory, because most applications have a limited number of data points they can operate on in parallel. This number can range from the single digits to the millions, but unless a SIMD architecture wants to limit the range of programs that it can usefully run, it cannot increase its SIMD width arbitrarily.

This simple mechanism for increasing performance with each new technology node meant that array and vector based SIMD architectures traditionally eschewed the use of caches or other auxiliary structures which would have used up area which could have been devoted to the SIMD data path.

While SIMD architectures have not been used by mainstream CPU manufacturers, they

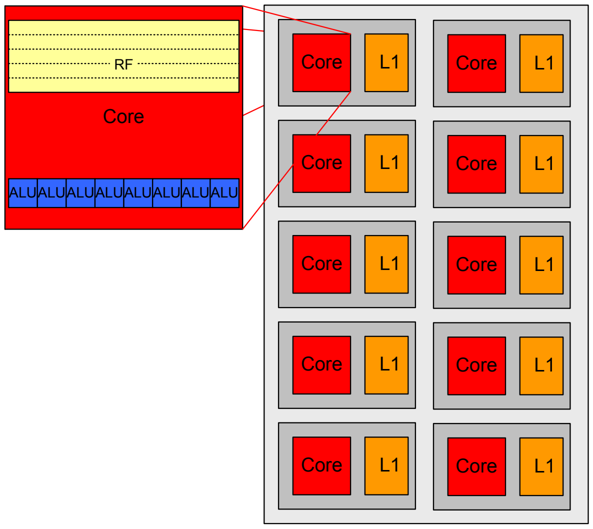

Figure 1.2: An illustration of current GPU architectures. GPUs contain many small cores, each with its own private caches (only one is shown for simplicity). Each core has multiple ALU lanes, which execute instructions in lockstep. The register file of each core is large enough to hold the register state of many threads. When a SIMD group (called a warp) stalls on a long latency operation, a hardware scheduler can mask this latency by continuing to execute other warps.

have been adopted by the manufacturers of graphics processing units (GPUs). I will use the example of GPU architecture

1.5 GPUs as an Example of Modern SIMD Architectures

Graphics processing units (GPUs) were once fixed-function hardware for 3D rendering. Demand for increasing programmability for such applications have gradually driven GPU architectures to become general-purpose manycore architectures (embedded within a system-on-chip including various 3D-specific accelerators). The introduction of hardware and software support for general-purpose programming languages on the GPU [19, 77, 80] has allowed GPUs to become a viable platform for general-purpose computing.

GPUs are optimized to provide high-throughput and to tolerate frequent long-latency accesses to graphics memory. This is due to the nature of the graphics workload that inherently has a very large number of independent tasks (hundreds of thousands of triangles and millions of pixels per rendered frame) and data access patterns with little temporal locality. As a consequence, GPUs have adopted an architecture similar to the MTA Tera [5]. As shown in Figure 1.2, each core is heavily multi-threaded and scheduling hardware decides each cycle which of the many threads to execute. This is necessary since threads frequently stall due to accesses to graphics memory, and many threads are needed to keep the ALU unit of a core reasonably occupied. In addition, each core uses a SIMD execution model and executes multiple threads in a single clock cycle. This organization is again a consequence of the graphics workload, where nearby tasks execute the same program (called shaders) and there is minimal control-flow divergence between tasks which execute the same program. A SIMD organization amortizes the area and power overhead of a core's frontend over many backends, increasing the total computational power achievable within a given power and area envelope. Note that SIMD lanes are referred to as threads and SIMD groups as warps or wave fronts in GPU terminology. I will use the terms thread and warp throughout this chapter.

GPUs do have caches, although the are much smaller than the caches of CPUs. The question might be asked why GPUs have any caches for data at all, since they are optimized to tolerate latency. The answer is that GPU caches are mostly meant as bandwidth savers and not as a way to decrease latency of memory accesses.

1.6 Tradeoffs between Multithreading and Cache Size

As noted above, GPUs employ heavy multi-threading as a way to tolerate memory latency. When adding traditional caches, which in this context are defined as supporting both reads and writes and having an access latency substantially lower than memory latency, there

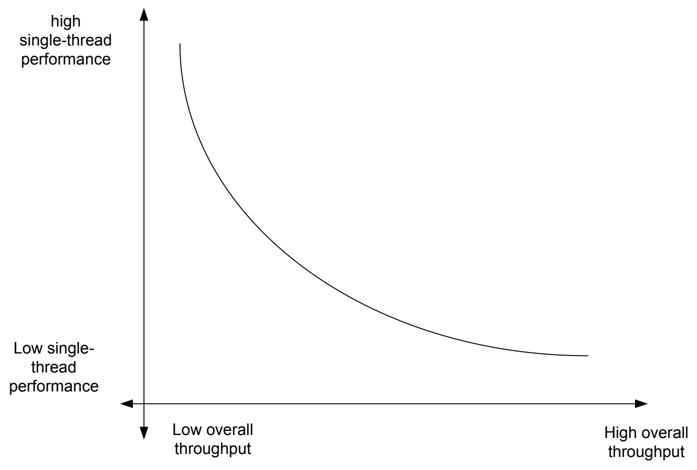

Figure 1.3: Tradeoffs in choosing core types

is an interesting balance between number of warps per core and the size of the per-core caches. More warps per core increase memory latency tolerance and performance, while increasing cache size for a given warp count will increase hit rate, decrease average memory latency and increase performance. But there is the problem that for a given cache size increasing the number of warps per core will put more pressure on the cache, sometimes leading to cache thrashing and a sudden jump in the required off-chip bandwidth and a *decrease* in performance.

While more warps per core increases performance, it also increases the size of the register file to hold the larger number of threads, as well as potentially requiring an increase in the size of caches to prevent “thrashing”. The best performance per unit of area is not necessarily achieved with the maximum number of warps per and the largest cache as I will show in Section 6.5.

1.7 Contributions of this Dissertation

To run the widest range of programs well, an asymmetric CMP must choose the right balance between the different types of cores, as well as optimize each component. Another issue is that, if there are large gaps between the different core types in terms of single-thread performance or number of threads they are capable of running, programmers will have a harder time getting the best performance form such a system under all circumstances. This point was illustrated by Marty and Hill [49], who showed that the ideal case for future CMPs would be if the hardware could fluidly reconfigure itself from running as a single, extremely capable single-threaded core to a very large number of simple processing elements *and all configurations in between*.

While such a system is clearly infeasible, it is clear that asymmetric CMPs will have different core types distributed along a curve, as illustrated in Figure 1.3, with the left side being the large and fast cores, the middle being made up of small throughput oriented scalar cores and the right side of the curve being made up of SIMD cores with low single-thread performance but maximum throughput and area efficiency.

This dissertation focuses on optimizing the architecture of throughput-oriented scalar and SIMD cores, which are the middle and left side of the conceptual curve.

Chapter 2 gives an overview of related work in the areas I explore, and explains how they relate to my own work.

I then focus on the gap in performance between the large, high-performance cores and the throughput-oriented scalar cores. For optimal performance across the widest range of programs possible, the gap should be as small as possible. But making the throughput-oriented cores faster, by replacing them with small out-of-order cores, might use too much area and sacrifice too much throughput. I show in Chapter 3 that out-of-order execution for small cores can be much cheaper than previously imagined, so that it might make sense to have slightly large, but much more capable MIMD cores, reducing the gap in single-thread performance between the large cores and the throughput MIMD cores. The fundamental insight in this chapter is that the traditional circuit structures needed for speculative out-of-order execution are overdesigned and not used efficiently during the common case.

In Chapter 4 I show another possible solution, by combining two multithreaded, in-order, scalar cores dynamically at runtime into a larger and faster out-of-order core. Both of these solutions allow an asymmetric CMP to have robust performance for a wider range of active threads. The key insight of my proposed solution is that the large register files of multithreaded cores can be repurposed to hold the re-order buffers of an out-of-order core and that, by not aiming for a very high-performance core, the overheads of a distributed core can be kept within acceptable bounds.

After having focused on the middle part of the curve in the first two chapters, I then investigate the memory performance bottlenecks of SIMD cores, which have turned out to be the real limiters for SIMD cores.

In my proposal for this dissertation, I promised to investigate ways in which SIMD cores could split SIMD groups at runtime, to increase their performance when the control-flow of threads in a SIMD group diverged. In my work on this topic, it became obvious that control-flow divergence was not the main limiting factor for many programs, but that it was memory divergence. Memory divergence occurs when all threads in a SIMD group execute a load or store, and a subset of them miss in a given cache level. This forces all threads in that SIMD group to wait until the all the threads which missed in the cache have received their memory values. In the worst case, a single miss can force a whole SIMD group to stall, slowing down execution of all threads. I thus focused on on solving the problem of memory divergence instead of control-flow divergence, since it had a much higher potential impact on performance for a wide variety of programs.

In Chapter 5 I develop a mechanism called diverge on miss, which allows SIMD groups to continue executing, even if a subset of their threads are waiting on memory. This mechanism greatly speeds up programs where memory accesses from a SIMD group are not always to contiguous addresses and a subset of the warp may miss in the data cache, decreasing the number of thread contexts needed to support a given level of performance or increasing the performance when holding the number constant. The key insight of diverge on miss is that SIMD cores which already support control-flow divergence and scatter/gather loads and stores already have most of the hardware needed to support a form of execution where threads can be at different points in their execution due to some having missed in the cache while others did not.

I also promised to investigate smart cache replacement and insertion policies for SIMD cores. My work on this topic showed that, contrary to my intuition, the performance benefit of such techniques was minimal. This was due to the fact that programs for SIMD cores that I investigated had inherently large working sets with either streaming behavior (no temporal locality) or only reuse between close by threads. The cache of each SIMD core was simply too small to capture any reasonable working set, even in the best case. Any smarter insertion or replacement policy could only improve hit rates marginally. Motivated by these insights, I investigated how a processor with many cores, each with relatively small caches, could facilitate reuse of data between the cores. While one possible solution to this problem would be to have a large shared cache, such an organization would have lower area-efficiency. While there have been prior proposals of how to use cache coherency protocols to deal with similar problems (see Section 2.5), I focused on the case without cache coherence. This case is particularly interesting because some manycore architectures (such as GPUs [68, 74]) do not support cache coherency, and because cache coherency can be particularly expensive for manycore architectures, which motivates looking at alternative solutions.

In Chapter 6 I introduce a way for non-coherent caches of SIMD cores to achieve most of the bandwidth and latency benefits available with a large shared last level cache, but at a fraction of the area overhead. The key insight in this chapter is that for non-coherent caches, tracking which caches contain which cache lines does not have to be precise or up-to-date. The directory structure of a directory-based cache coherence protocol becomes a predictor and a lookup a mere performance hint. Erroneous predictions may reduce performance but do not violate memory semantics.

Together, the techniques described in this dissertation improve the performance, power- and area-efficiency of small, throughput-oriented cores of different types. They increase the set of workloads which can profit from SIMD cores and show how to build CMPs which can dynamically adapt their throughput cores to workloads with fewer threads.

Chapter 2

Related Work

2.1 Power-Efficient Out-of-Order Structures

The power consumption of a modern speculative out-of-order core is dominated by the power used by the large and complex buffer structures required for out-of-order execution. Chief among these are the issue queue and load/store queue. Since modern processors are primarily limited by their power consumption there has been substantial effort devoted to finding lower power organizations for these structures.

Sethumadhavan *et al.* [98] explore the problem of scaling a traditional LSQ design to larger sizes in terms of access delay, while Gandhi *et al.* [40] explore scalable and power-efficient alternatives to traditional LSQs in the context of a core which can speculatively execute thousands of instructions. Both works found that only a small fraction of loads have values forwarded to them by stores, and design their scalable LSQs around this fact. The Memory Alias Table (MAT) I present in Chapter 3 builds on some of the ideas about bloom filters presented in these papers.

Buyuktosunoglu *et al.* [23] examined the power consumption of issue queues and the scheduling logic in real high-performance processors and proposed an instruction scheduling mechanism which did not rely on the instructions being strictly age-ordered inside the issue queue. Sassone *et al.* [94] also explored the scalability of traditional issue queues, and proposed improvements on the alternative matrix scheduler [45] for lower power consumption and better scalability. Huang *et al.* [53] looked at a hybrid issue logic, which combined broadcast and direct lookup for lower power consumption.

The consumer-based issue queue I present in Chapter 3 draws its inspiration from Huang, but completely abandons the need for broadcast and match logic. I also use the findings about pseudo-random scheduling from Sassone and Buyuktosunoglu for a further simplification of my design.

The Store Vulnerability Window (SVW) was introduced by Roth [89] as a verification mechanism for load/store aliasing and ordering which could be used in conjunction with several load speculation techniques. The Memory Alias Table is a similar structure to the SVW, but uses much less hardware. More recent work [101, 111, 100] has tried to largely or completely eliminate the Load-Store Queue (LSQ) by using the SVW as the checking mechanism for speculative forwarding, which we avoid due to its complexity. Our work differs in that we do not try to replace a part of an OOO processor, but instead augment a simple in-order processor so that it can detect memory order violations with minimal hardware cost. We also do no speculative forwarding; indeed, we abandon forwarding completely in our design.

2.2 Combining Cores

There has been much recent interest in how to combine multiple small cores to execute a single-threaded program faster. The most influential work in this area is no doubt the Multiscalar architecture from Wisconsin [106], which used a combination of hardware and software to execute normal sequential programs on a number of in-order cores connected by a ring.

More recently, work on combining several smaller cores into a single larger and more capable core was performed by İpek *et al.* [55]. We compare this work to Federation in detail in Section 2.2.1.

The Voltron architecture from Zhong *et al.* [121] allows multiple in-order VLIW cores of a chip multiprocessor (CMP) to combine into a larger VLIW core. It requires a special compiler to transform programs into a form which can be exploited by this larger core. The performance is heavily dependent on the quality of the code the compiler generates, as the hardware cannot extract fine-grained instruction parallelism from the instruction stream by itself. The work on Composable Lightweight Processor (CLP) [63] leverages the block-level dataflow EDGE ISA [21] and associated compiler [105] to allow small cores to work on a single instruction stream, without having to use traditional out-of-order structures such as a rename table or issue queue. Our work does not assume an advanced compiler and is applicable to RISC, CISC, and VLIW ISAs.

Federation, the technique I introduce in Chapter 4, differs from Voltron and CLP in that it does not require a special ISA, but can instead use any ISA. Federation differs from Core Fusion in that it does not assume that the underlying cores are already out-of-order, but instead adds all the necessary out-of-order structures. The main takeaway from Core Fusion is that constructing a very wide out-of-order core from small cores faces the issue of steeply diminishing returns in terms of the amount of performance gained for extra hardware added. This is chiefly due to the extra latency introduced by the extra interconnect and the known performance limitations of very wide out-of-order cores when not paired with almost perfect branch prediction and memory bypass prediction. In Federation I tried to minimize any extra latency added to the pipeline and avoided building interconnects which pass over multiple cores.

Salverda and Zilles [93] explore the performance limits of a design that contains a number of in-order lanes or pipelines that can be fused at run-time to achieve out-of-order execution. Their work assumes a “slip-oriented out-of-order execution model,” in which out-of-order execution only occurs when the individual lanes slip with respect to one another. In other words, within each lane, instructions always execute in-order. The performance constraints shown in their work are only valid for machines that utilize this execution model. Federation is *not* based on the slip-oriented out-of-order execution model. When in-order pipelines are federated, instructions can be issued out-of-order to any pipeline and thus instructions within the same pipeline can execute out-of-order with respect to one another. This approach raises some scaling issues of its own but frees Federation from the fundamental constraints of the slip-oriented model.

Work by Kumar *et al.* [67] on heterogeneous cores showed that the overall throughput of a heterogeneous CMP can be higher than an area- equivalent homogeneous CMP, if the OS can schedule threads to different types of cores depending on their needs. However, because the mix of large and small cores has to be set at design- time, the OS or hypervisor cannot dynamically make a tradeoff at runtime between the number of cores (i.e., the number of thread contexts) and single- thread latency. Grochowski *et al.* [46] follow up on this line of work and observe that the combination of performance- and throughput- oriented cores with dynamic voltage scaling can provide a better combination of single-thread latency and throughput than either technique can provide alone.

Adjoining cores which are federated have their caches merged when in federated mode, similar to [66, 34]. However, we do not require two cores to be able to access the same cache simultaneously, since only one core’s load and store ports are active when in federated mode.

Numerous groups have evaluated various combinations of clustered OOO processors and multi-threading. İpek *et al.* [55] provide a comprehensive overview of this body of work. Another approach to improve the single-thread performance is to use runahead execution [30, 78], which is orthogonal and even complementary to federating two simple cores. Runahead reduces time spent waiting on cache misses, which would potentially help the more powerful federated core relative to the underlying scalar core. Additionally, federating two cores would help the runahead thread run faster and thus further ahead of the main thread.

2.2.1 Comparison to Core Fusion

The work on Core Fusion [55] provides an interesting comparison point to Federation. Core Fusion and Federation employ very different approaches to the problem of how to aggregate smaller cores into a single, higher performance core. Core Fusion aims to build an OOO core with a very deep execution window and lots of execution resources. To achieve this goal, Core Fusion combines a larger number of cores (up to four cores) than Federation (two cores). Due to the complexity of the extra structures needed for Core Fusion and the latency required to communicate between several cores at multiple locations in the pipeline, Core Fusion must increase the length of many of the critical loops of the processor pipeline. Federation employs almost exactly the opposite approach, focusing on aggregating fewer, smaller cores and placing an emphasis on NOT increasing any of the critical loops of the pipeline unless absolutely necessary. The choice of a centralized Issue Queue and centralized MAT stem directly from trying to avoid such overheads. We believe that the large body of work on clustered architectures show convincingly that distributing the critical structures of an OOO core only makes sense if the workload exhibits large amounts of ILP and few serializing conditions such as branch mispredictions and memory aliasing events; conditions which are not true for many applications which are not easily decomposed into multiple threads and thus need higher single-thread performance the most.

In both [55] and in this paper, the performance of the aggregated core is compared to that of a dedicated 4-way OOO core. Of course, directly comparing the results is difficult, since the dedicated cores in the two comparisons are configured differently and use different simulation methodologies. Nevertheless, Core Fusion of four 2-way OOO cores achieves about 102% and 115% of the performance of a dedicated 4-way core on SpecInt and SpecFP, respectively. We show in Section 4.5.3 that Federation of two 2-way in-order cores achieves 88% and 95%, respectively, of the performance of a dedicated 4-way OOO core, with half the execution resources and much lower power. Thus, even with much simpler baseline cores, Federation is able to achieve performance that is competitive with Core Fusion.

Comparing the areas of the aggregated cores is not necessarily useful, since one can assume that a manycore processor will have more than enough cores for any aggregation technique. Comparing the area overhead of the aggregation techniques and the area efficiencies of the baseline cores is more instructive. The area overhead of Core Fusion is estimated to be 8.64mm^2 from a 200mm^2 die with 100mm^2 devoted to core area, or about 8.6% of the core area. Using scalar in-order cores as a baseline, we estimate in Section 4.4.2 that the area overhead of Federation is approximately 3.7% of each pair of cores, and thus 3.7% of the total core area regardless of the number of cores. Using 2-way in-order cores with branch prediction as a baseline, the area overhead is much smaller, since the majority of the area overhead of federating scalar cores was due to the addition of a small branch predictor.

For phases of execution in which the thread count is high, a manycore processor implementing either Core Fusion or Federation will be best off without any cores fused/federated in order to provide as many hardware thread contexts as possible. In such a case, Federation's multi-threaded in-order baseline cores will provide much higher aggregate throughput than Core Fusion's 2-way OOO baseline cores because of their significantly higher area efficiency. Carmean [25] estimates that a multi-threaded in-order core takes up only one-fifth the area of a traditional core while providing more than 20 times the throughput per unit area. Thus, Core Fusion will provide superior performance when the thread count is extremely low. For medium to high thread counts, however, the higher throughput of the underlying cores in Federation will provide significantly higher performance.

2.3 SIMD Hardware

The use of SIMD instruction sets and hardware was first proposed for early supercomputers aimed at scientific applications, examples of which include the Illiac IV [15], Cray-1 [90] and Connection Machine [119]. SIMD instructions were used to replace a large number of loop iterations with a small number of SIMD instructions working on a large sets of data.

Modern microprocessors adopted a very limited form of the SIMD model by adding short vector extensions to existing ISAs [64, 82, 85, 116]. These extensions were primarily aimed at audio, video and graphics processing and are limited to vector length of four 32-bit elements.

GPUs have traditionally used SIMD execution, since their execution model was to execute the same small program (called a shader) on a very large number of inputs [68, 74].

2.4 Diverge on Miss

Early academic work [11, 83] on manycore processors explored the benefit of chips built out of many simple cores for both commercial and scientific workloads. They showed that for workloads with enough parallelism many simple cores could outperform large, few high-performance cores. Recent commercial, general-purpose products that target throughput-oriented workloads exemplify some of these lessons. For example, the Niagara processor [3] from Sun implements 8 simple SPARC cores, each of which has 4 execution contexts.

GPU manufacturers have evolved their designs from being pure ASICs to manycore processors, with each core having a number of logical warp width between 32 and 64 and a large number of warps per core [68, 7].

While all of this hardware was traditionally hidden behind complex graphics APIs, recently both AMD and NVIDIA have made available APIs [43, 19, 6] which are meant for general purpose computation and can take advantage of GPU hardware.

The recently announced Intel Larrabee architecture [97] has the capabilities of both GPUs and multicore processors, supporting both the x86 ISA, cache coherency and memory ordering, as well as wide SIMD execution and multiple hardware execution contexts per core. Both Niagara and Larrabee (will) support conventional cache architectures, where caches are coherent, are addressed through a unified address space, obey a well-defined memory ordering model and are large enough to hold the working set of many programs.

GPUs on the other hand, because they have been designed primarily to support graphics APIs such as OpenGL and Direct3D [13], have very different cache architectures. One primary difference is simply in the size of caches relative to number of ALUs. Another difference is that caches are divided among different address spaces (so called texture and constant caches) and optimized for specific access patterns which go along with these address spaces in graphics applications. Although a variety of CUDA applications have taken advantage of these properties, Che *et al.* [28] and Boyer *et al.* [16] in particular discuss the importance of using these memory paths.

In general it can be said that GPUs have designed their cache architectures to help maximize aggregate throughput, but not necessarily to minimize the latency of any individual thread. Slipping warps enable the combination of very wide SIMD execution of GPUs with regular cache hierarchies and help greatly reduce single-thread latency and increase throughput for workloads with irregular access patterns.

Warp divergence in SIMD processors as a result of control-flow divergence was explored by Fung *et al.* [39], who proposed Dynamic Warp Formation as a way to lessen the performance loss due to this particular problem. While the technique of dynamic warp formation can also be applied to memory divergence, the hardware overhead of our technique is much smaller, requiring only small additions to existing structures. For example, Dynamic Warp Formation requires that the register file have as many independent banks as there are threads in a warp, substantially increasing the area overhead due to addresses having to be routed to each bank, each bank needing its own address decoders and also having much shorter word lines. Our technique requires only one bank for the width of the warp.

2.4.1 Comparison to Architectures with Scratchpad Memories

The software controlled approach to diverge on miss outlined in Section 5.3.2 can be compared to the streaming approach of the Merrimac architecture [31] and the Cell chip's Synergistic Processing Units [38]. These architectures have explicit memory hierarchies and independent DMA engines, which can fetch lists of memory references into a large software controlled on-chip buffer asynchronously, without having to block execution.

In contrast to these architectures, a software implementation of diverge on miss does not force the programmer to explicitly organize data in a fixed size buffer, nor does it fix the size of this buffer. Any program written for a von Neumann architecture will work on such a processor. The extra instructions to snoop the memory hierarchy only provide potentially higher performance.

2.4.2 Comparison to Warp Subdivision

In our prior work by Meng*et al.* [72], on which I was a co-author, he proposed a hardware technique called dynamic warp subdivision to deal with the problem of warps stalling due to divergent memory accesses in a SIMD core. He proposed to move threads which have hit the cache on a given access to a new warp (a so called warp split), which can be scheduled and executed independently from the parent warp, while leaving the register contents of the affected threads in place.

The main limitation of warp subdivision is that it allows only a very limited number warp splits (due to the extra scheduler entry needed for every warp split) per warp, usually 1. The problem with this approach is that threads can miss and hit in the cache repeatedly in an unpredictable pattern, and that a split warp will still have to stall just like a normal warp would if any of its threads misses the cache.

The key insight behind diverge on miss is that slipping warps can continue to execute even if threads repeatedly miss the cache and is not affected by the pattern of cache misses between threads in the warp. On the hardware side, the drawback of warp subdivision is that the warp scheduler needs to be doubled in size compared to the baseline architecture. This can affect the critical path of the core, either decreasing frequency or making back-to-back execution of the same warp impossible, decreasing single-warp performance.

2.5 Sharing Tracker

As an alternative to hardware cache coherence, which poses a number of design challenges, software-controlled coherence has been proposed as a more scalable and lower-cost solution for cc-NUMA and virtual distributed shared memory (VDSM) multicomputer organizations. A simple version of software coherence is for the programmer to manually flush caches when switching between reading and writing, or to double buffer, with separate (cached) input and (uncached) output data structures. This does not present a great burden when the sharing is infrequent and occurs in well-defined patterns. In order to support finer-grained sharing, considerable work was done in the 80s and 90s to enable the compiler to automatically manage coherence in hardware shared-memory systems [29, 108] and to reduce the cost of network transactions for VDSM. In the case of VDSM, the main techniques were to reduce the frequency and size of updates (e.g. Munin [26]) and reduce the latency of those updates (e.g. Shrimp [12]). These techniques generally required operating system support (to manage shared pages) and potentially hardware support (new network interfaces).

Chip multiprocessors have an advantage in this regard, because sharing can be managed natively in hardware and all cores share a common pool of global memory. Other multicore organizations take advantage of this to eschew hardware coherence, e.g. RAW [120] and Cell [58]. GPUs take advantage of shared global memory to optimize the L1 caches for data that is read-only or exhibits only coarse-grained sharing. Although details differ, GPU architectures from NVIDIA [68] and AMD [74] both support similar memory hierarchies; for more details, see the next section. Briefly, fine-grained read-write sharing and synchronization objects are expected to be localized into the PBSM (per-block scratchpad) or accessed only through global memory. Deep multithreading allows other threads to hide latency of threads stalled on global-memory access.

Bakhoda *et al.* [10] evaluate a multi-level, hardware-coherent cache hierarchy for GPUs but results are inconclusive. Our work proposes an alternative that avoids the challenges of hardware coherence.

A huge body of work has of course explored the more conventional alternative of supporting hardware cache coherence in multicore organizations and various optimizations that can be built on top of a coherent organization. We briefly mention work that we believe is most closely related to our line of investigation.

Chang *et al.* [27] use *cooperative caching* to share the resources of a number of private caches on a single chip. They use a *central coherence engine* which replicates the tags of all private caches. Requests which miss in core's private L2 cache access the coherence engine to check whether the requested cache line is in an L2 of a different core. They also add mechanisms for intelligently replicating cache lines and having evicted cache lines spill to another on-chip cache. The drawback of their technique is that each request needs to check a large number of tag arrays (as many as there are cores on the chip minus one), which is a power-hungry process, and that a single cache line can have copies in multiple L2s, which wastes space in the coherence engine.

Herrero *et al.* [47] build on cooperative caching with their work on *distributed cooperative caching*. They replace the replicated L2 tag arrays of the central coherence engine with a distributed, address-indexed tag array, reducing the number of tag comparisons any request has to make to determine whether a copy of its requested cache line exists somewhere on chip. Our work differs in that the sharing tracker is not a full coherency directory, substantially reducing the required hardware and eliminating the complexity of traditional coherence hardware.

Destination Set Prediction [70] assumes a cache-coherent multi-processor where each core has its own L2 cache, and each L2 has its own predictor, which predicts which other core/L2 cache has current ownership of certain cache lines. Destination set prediction was designed for workloads with low degrees of sharing between cores, such as commercial workloads. Our sharing tracker differs from destination set prediction as it is useful for workloads where there can be large degrees of sharing of cache lines with irregular patterns. The sharing tracker tracks cache line information at the global level, while destination set prediction keeps track of which other cores a given core previously has exchanged cache lines with.

There has also been considerable work on caches with non-uniform access latencies [62, 54] (so called NUCA caches). NUCA caches are built from a large number of memory tiles, which are addressed by a smart controller, which can move around cache lines based on recency of access or alter the degree to which a memory tile is shared between cores. NUCA caches take a fundamentally different approach from our own, since they focus on intelligently mapping cache lines based on address or giving cores a fixed and uniform amount of sharing with a given set of other cores. The sharing tracker is purely demand driven only restricted by the capacity and associativity of the caches it covers and does not restrict sharing between any core anywhere on the chip.

For *modeling* GPUs, Bakhoda *et al.* [10]'s simulator *GPGPUSIM* is an execution driven simulator which can run kernels compiled to NVIDIA's PTX assembly format and closely models a current generation NVIDIA GPU. Our simulator takes a fundamentally different approach, by instrumenting data-parallel applications and collecting only their data access traces. Our simulation approach is discussed further in Section 6.4

Chapter 3

Lightweight Out-of-Order Execution

3.1 CMP Architecture Tradeoffs

CMP designers face the difficult task of what combination of single-thread performance and overall chip throughput to target. Single-thread performance is improved by using large, complex and power-hungry cores. Overall throughput is maximized by going with a large number of simple cores, which are small and use little power. The current trend is to choose design points which either have a small number of large cores [35, 59, 71, 91, 107, 109], or a large number of small cores [3, 56, 97]. Those designs that opted for small cores have all forgone the use of out-of-order execution for their chosen cores, judging it to not be area and power efficient enough compared to adding more thread contexts per core. I think that out-of-order execution is judged as being not area- and power-efficient primarily because it has been used in the past for cores which had performance as their primary design goal, which meant that they were willing to use area less efficiently if it meant higher performance.

I think that out-of-order execution can be a good design choice for designs which want high throughput, but also want good single-thread performance. I achieve this by re-engineering the major hardware structures required for out-of-order execution for much lower complexity and power by replacing content addressable memories (CAMs) and broadcast networks with simple lookup tables. This makes it possible for manycore processors to offer competitive single-thread performance without incurring the major area and power hit of a dedicated fast core.

3.2 Minimal Branch Prediction

Branch prediction is implemented using Next Line and Set prediction (NLS) [24, 61, 117] instead of a branch target buffer. NLS maintains an untagged table indexed by the branch address, with each entry pointing to a line in the instruction cache predicted as the next cache line to be fetched. NLS predicts the location in the cache where the next line will be fetched rather than the actual address to be fetched. This significantly reduces the overhead of supporting NLS. For example, implementing a 512 entry NLS requires only about 0.75 KB of extra state. A small return address stack (RAS) is also added, which requires only 256 bits of state. I have omitted the top 32 bits of the return addresses and assume they do not change. No negative performance impact on our workload is noticeable from this simplification.

An overhead of speculative OOO execution that is often overlooked is the fact that the rename table of an OOO core has to have some way to recover from branch mispredictions [104]. The usual way is for the rename table to be checkpointed at each branch, and the checkpoint of a branch restored when it is detected that that branch was mispredicted.

For the lightweight core branches are resolved at commit time, obviating the need to maintain multiple snapshots of the speculative rename table or the need to walk the active list (AL) in the case of a branch misprediction. The core simply needs to maintain two copies of the rename table, a speculative version, updated in the rename stage, and a non-speculative version updated in the commit stage. If a branch misprediction is detected, the core waits for the branch to reach the commit stage and be the oldest instruction in the active list. The non-speculative rename table is then simply swapped with the speculative version and execution can continue. Note that swapping the contents can be accomplished by keeping the speculative and non-speculative version in the same physical SRAM or latch array structure, with one version occupying the upper half of the structure and the other the lower half. A simple bitvector (one bit per architected register) can then indicate whether the version to read out is in the upper or lower half. The bitvector is reset when a mispredicted branch reaches the commit stage, and each bit is set when an instruction flowing through the rename stage writes to the corresponding architected bit.

I have also explored the performance impact of limiting the number of branch checkpoints. However, since my focus is on simplicity, my base case for the lightweight core still uses commit time branch recovery, which reduces performance by approximately 5%. Adding just two snapshots of the rename table would almost completely eliminate this overhead, but I show results for the simplest case.

3.3 Consumer-Based Issue Queue

The area and power constraints of our design prevent the implementation of a traditional CAM-based Issue Queue (IQ). To avoid tag broadcast or tag match logic, I use a simple table in which consumers "subscribe" to their producers by writing their IQ position into one of the producer's IQ entry's consumer fields, similar to a ideas by Huang *et al.* [53] , Brekelbaum *et al.* [17] and Sato [95]. Huang *et al.* add a subscription mechanism to a traditional broadcast-based IQ, and limit the number of subscribers to one. Brekelbaum *et al.* use a consumer-based IQ as a large L2 IQ, where producer selection and consumer wakeup are decoupled, unlike my design, which uses no broadcast hardware and directly wakes up consumers. Sassone *et al.* [94] showed that, for a processor with a 96-entry instruction window, over 90% of all dynamic instructions have no more than one dependent instruction in the instruction window when they execute. Thus, each IQ entry in our design only has a small number of consumer fields. The exact number of consumer fields per entry is a design choice; we found that limiting the number of fields per entry to two reduced performance by only a fraction of percent compared to a traditional IQ. This performance impact is evaluated in greater detail in Section 4.4.

Each entry in the IQ holds the usual opcode, register ids and immediates, but also has several consumer id fields and two ready bits, which are set when the left and right operands become available, respectively. On issue, each instruction checks its consumer fields and sets the appropriate ready bits in the consumer's entry. If both input operands are ready, the ready signal for that entry is sent to the scheduler.[1] Each entry in the issue queue also requires two fields for the active list IDs of the producers of its input operands, a field to store its opcode, and a field to store its immediate/displacement value. These extra fields are not required for the critical wakeup and select loop and can thus be stored in a table physically separate from the ready bits and the consumer IDs.

Since the number of consumer fields is small, an instruction can stall if its producer is oversubscribed. This necessitates the addition of an extra bit to each producer entry in the IQ which is set if the producer is oversubscribed. If this bit is set when the instruction executes, a signal is sent to the rename stage to unstall the dependent instruction(s).

The normal scheduling logic for an out-of-order processor tries to issue older instructions first. It is usually complex and power hungry. I instead implement a simpler pseudo-random scheduler [94] which uses a static priority encoder and does not take into account the age of different instructions. For a small out-of-order window, this simplified scheduler reduces performance by around 1%.

[1]Note that all loads can issue speculatively, without waiting on unresolved stores; see Section 3.4 for an explanation.

3.4 Replacing the Load/Store Queue with the Memory Alias Table

Traditional Load-Store Queues (LSQs) are used for enforcing correct ordering between loads and stores which can potentially execute out of program order, and to forward values between aliasing loads and stores. They have large CAMs for address matching, circuitry for age prioritization in case of multiple matches, and a forwarding network. All these structures would add considerable power and complexity to our baseline processor. Instead, I propose the Memory Alias Table (MAT), which builds on ideas from the Store Vulnerability Window (SVW) [89] and work by Garg *et al.* [42]. Contrary to this previous work, the MAT only detects memory order violations and does not provide a mechanism for forwarding store results to younger loads, eliminating the need for a forwarding network which can deal with multiple (partial) matches. Since previous work has shown that store-to-load forwarding is rare even in large OOO cores [89, 42], omitting the forwarding network provides considerable area savings with minimal performance loss.

Memory order violations must be treated as branch mispredictions and re-executed. Unlike in [101], I do not implement a load-store alias predictor, but statically predict all loads and stores to not alias. A dynamic predictor is necessary for a large, high-performance design, where accurate store-to-load forwarding is needed to exploit the available machine resources, but can be omitted from our small design.

3.4.1 Concept of the Memory Alias Table

Before explaining the operation of the MAT, I should clarify our usage of certain terms. When discussing program order, I refer to instructions as earlier or later; when discussing actual execution order, I refer to instructions as younger or older.

Conceptually, the MAT operates as follows: each load places a token in an address-indexed hash table, which is removed (becomes invalid) when the load commits. Each store checks the hash table at commit for a token from a younger load which is still in the pipeline. Any store finding a valid token when it is committing knows that the token is from a later load and signals a memory order violation. The store does not need to cause an immediate pipeline flush but instead leaves an exception token in the table when it commits. The offending load will discover this exception token during commit when it invalidates its token in the hash table. The load can then either replay or cause a pipeline flush.

The hash table proposed by Garg *et al.* [42] utilizes the same basic concept as the MAT, while the SVW inverts the relationship between loads and stores, with stores leaving tokens in a table and loads checking the table for valid aliasing entries. A critical distinction between the MAT and these previous proposals is how instruction age is represented in hardware. Previous proposals used a store sequence number (SSN) or a load sequence number (LSN) to determine relative age. Since it is non-trivial to determine when the last load vulnerable to a store committed, a counter representing dynamic instruction age was used. This required relatively large entries and the comparison of 16-bit or larger values to determine the relative ages of a load and a store. Other proposals [98] used simple counting bloom filters, but could not determine the relative age of a load or store.

The MAT uses a simpler approach: each load increments a simple counter when it executes and decrements the same counter when it commits. Stores check the MAT only when they commit. Since any earlier load will have removed any sign of its presence from the MAT before a store reaches commit, the store knows that if its counter in the MAT is non-zero, there must be at least one later load in the pipeline with which it potentially aliases. (Previous proposals had the store check their equivalent of the MAT as soon as the address generation for the store was complete. They thus had no way of telling if an aliasing load was later than the store or not; they could only determine that it was older.)

Since our proposal relies on the precision of the counters in the MAT for correctness, the number of bits in each counter must equal the logarithm of the size of the AL. Note that because our design does not have a separate LSQ structure, the whole AL can be filled with loads and/or stores. Even for much larger instruction windows than I discuss here, the size of the counters is still much smaller than the 16 bits required to store the SSN[2] in [101]. Moreover, multiple counters can share a single set of higher-order bits (with only the LSB private to each counter), further reducing the amount of storage required per entry. The sharing of the upper bits can be considered the inverse of sharing the LSB in certain branch predictor tables [99]. I show in Section 4.4 that sharing all but the LSB between multiple counters is a feasible approach, as it introduces very few extra false positive memory order violations.

3.4.2 Dealing with Coherence and Consistency

To enforce a memory consistency model in the presence of cache coherence, the MAT must ensure that no load gets the wrong value, even if it initially executed out of program order. Two loads from the same location can be out of order with respect to each other as long as no change to that location occurs between the two accesses. To ensure this property, any cache coherence transaction indexes into the MAT and sets the exception bit for its entry or entries. Any load to this location which is in the window when this occurs will force a flush of the pipeline.

Any load committing in the same cycle as the cache coherency event can ignore it, since it is assured to have received its value before the event. Any committing load which decrements the counter to zero can reset the exception bit, since no loads which have

[2]The SSN can be smaller than 16 bits, but since overflowing the SSN requires a pipeline flush and a reset of the hash table, a smaller SSN leads to lower performance.

already received their values and have this location as their target are in the window any longer. To ensure forward progress, the first load to see the exception bit at commit can still commit, since it cannot have received the wrong value in any combination of events. This load can set a second bit (shared across the whole table), to indicate that later stores are not the first to have seen the exception bit. This bit is reset at all pipeline flushes.

The performance of the MAT, the SVW, and a traditional LSQ is compared in Section 4.4.

3.5 Simulation Setup

I evaluate our design using a simulator based on the SimpleScalar 3.0 framework [20] with Wattch extensions [18]. For the OOO cores, our simulator models separate integer and floating point issue queues, load-store queues and active lists. The pipeline has been expanded from the 5-stage pipeline of the baseline simulator to faithfully model the power and performance effects of the longer frontend pipelines. When simulating the MAT, our simulator allows loads to issue in the presence of unresolved stores. In the case that a memory order violation occurs, the pipeline is flushed when the offending load attempts to commit.

Wattch has been modified to model the correct power of the separately sized issue queues, load-store queues and active lists. Additionally, I accurately model the power of misspeculation in the active lists. Static power has been adjusted to be 25% of max power, which is closer to recently reported data [73].

I use the full SPEC2000 suite with reference inputs compiled for the Alpha instruction set. The Simpoint [102] toolkit was used to select representative 100 million instruction traces from the overall execution of all SPEC2000 benchmarks. For each run the simulator was warmed up for 10 million instruction before statistics were kept to avoid startup

Parameter	**2-way**	**4-way**
Active List	32	128
Issue Queue	16	32
Load-Store Queue	16	64
Data Cache	16KB	32KB
Instruction Cache	32KB	32KB
Unified L2 Cache	256KB	2MB
Branch Target Buffer	512	4K
Direction Predictor	2K bimodal	16K tour.
Memory	100 Cycles, 64-Bit	

Table 3.1: Simulator parameters for the different core types. The lightweight core has the same sized resources as the dedicated 2-way core. Note that the lightweight core use an MAT instead of an LSQ, and thus the number of loads and stores is limited by the size of the Active List rather than the size of the LSQ.

effects. When presenting averages across the entire benchmark suite, I weigh all benchmarks equally by first taking the average across the multiple reference inputs for those benchmarks that have them.

The lightweight core is compared against three other cores: a baseline scalar, in-order core; and traditional, dedicated 2-way and 4-way OOO cores. The simulation parameters for the different cores are listed in Table 3.1. Note that the smaller L2 for the small cores represents a single tile of a much larger L2, to simulate the fact that these cores will not be the only cores active on the chip and thus do not have exclusive use of the whole L2.[3]

3.5.1 Area Estimation Methodology

To show the area benefit of the lightweight core I need a way to calculate area numbers for different core types. Estimating the sizes of the different core types and the area overhead of the lightweight core is a difficult task, and I can only provide approximate answers without actually implementing most of the features of the different cores in a specific design

[3]I also simulated all cores with a 32MB L2 cache and verified that while absolute performance improves by about 20%, this occurs across the board, so that the relative performance between the lightweight and the dedicated OOO cores changes by less than 0.9%

Core Type	**Size in** mm^2
1-way in-order	1.739
Lightweight 2-way OOO	3.945
2-way OOO	5.067
4-way OOO	11.189

Table 3.2: Estimated sizes for core types in 45nm technology.

Subscriber Slots	**Change in IPC**
1	-0.71%
2	-0.34%
4	-0.04%
8	0.00%

Table 3.3: Impact on arithmetic mean IPC of the number of subscriber slots in the subscription-based IQ. The change in IPC is computed relative to a traditional IQ.

flow. To estimate realistic sizes for the different units of a core, I measured the sizes of the different functional units of an AMD Opteron processor in 130nm technology from a publicly available die photo. I could only account for about 70% of the total area, the rest being x86-specific, system level circuits, or unidentifiable. I scaled the functional unit areas to 45nm, assuming a 0.7 scaling factor per generation. The sizes of the different cores were then calculated from the areas of their constituent units, scaled with capacity and port numbers. These final area estimates are shown in Table 4.5.

3.6 Results

I first present results from sensitivity studies of the changes to the major structures introduced earlier. To isolate the performance impact of each feature and to avoid artifacts due to clustering, I evaluate each feature separately in the traditional, dedicated 2-way OOO core.

Table 3.3 shows that restricting the number of subscription slots in each IQ entry has very little impact on overall performance. I attribute this to the fact that the majority of

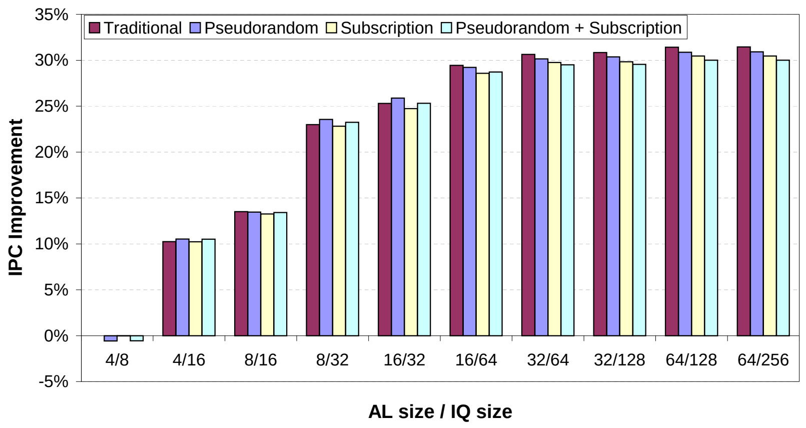

Figure 3.1: Increase in arithmetic mean IPC for different IQ designs as the sizes of the IQ and the AL are increased. By default, designs use oldest-first scheduling and a CAM-based IQ. Designs labeled “Pseudorandom” instead use pseudo-random scheduling and designs labeled “Subscription” instead use a subscription-based IQ. The percent improvement in IPC is in comparison to the “Traditional” configuration with a 4-entry IQ and an 8-entry AL.

dynamic instructions have only a single consumer [22], and that only a fraction of those consumers are in the IQ at the same time as their producers. Based on these results, each entry in the lightweight core has two subscription slots. Figure 3.1 shows the scaling behavior of the subscription-based IQ compared to a traditional IQ, as well as the impact of using pseudo-random scheduling instead of oldest-first scheduling. The impact of both changes is very small for all configurations, which is in agreement with previous work [94]. The largest combination of IQ and AL shows only a 1.1% difference in absolute performance between the best and worst configurations.

The use of the MAT allows most loads to execute earlier than they would have with a traditional LSQ, but at the cost of additional pipeline flushes due to both true memory order violations and false positives from the limited size of the hash table. Figure 3.2 shows the performance of the baseline core using either a MAT, SVW, or LSQ. As the sizes of of the hash tables are increased, the false positives are reduced and essentially only the true memory order violations remain. Note that since both the SVW and the MAT place no

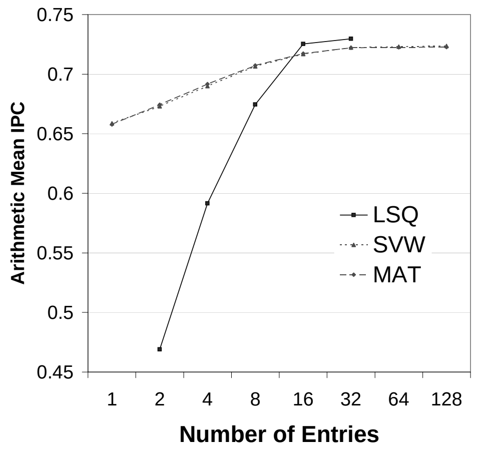

Figure 3.2: Scaling of arithmetic mean IPC for LSQ, SVW, and MAT as the number of entries is increased. The lines for the SVW and MAT are almost indistinguishable.

restrictions on the number of loads and stores in the pipeline, even a 1-entry SVW or MAT can have as many loads simultaneously in flight as there are AL entries. The MAT and SVW have almost exactly the same performance and both use much less hardware than the LSQ. As each entry of the SVW is 16 bits and each entry in the MAT is only 6 bits, the MAT provides the best performance for a given amount of hardware. Since I would need an 8-entry LSQ to outperform even the smallest MAT, the tradeoff of hardware overhead versus performance is a very favorable one.

As discussed in Section 3.4.1, the MAT can save even more hardware by sharing most bits of each counter among neighboring entries in the hash table. Table 3.4 shows the impact on performance as I increase the number of counters sharing one set of upper bits. While the performance impact is minimal, the numbers are noisy, since intuitively more sharing should produce more false positives in the hash table and therefore lower performance. For the lightweight core, I share one set of upper bits between eight entries, so each entry only uses $1 + \frac{4}{8}$ bits for the counter and an additional $\frac{1}{8}$ bit for the shared exception bit.

The overall performance of the lightweight core compared to scalar, traditional 2-way

Sharing Degree	**Change in IPC**
2	-0.46%
4	+0.02%
8	-0.05%
16	-0.18%

Table 3.4: Impact on arithmetic mean IPC of sharing the higher order bits of each counter in the MAT.

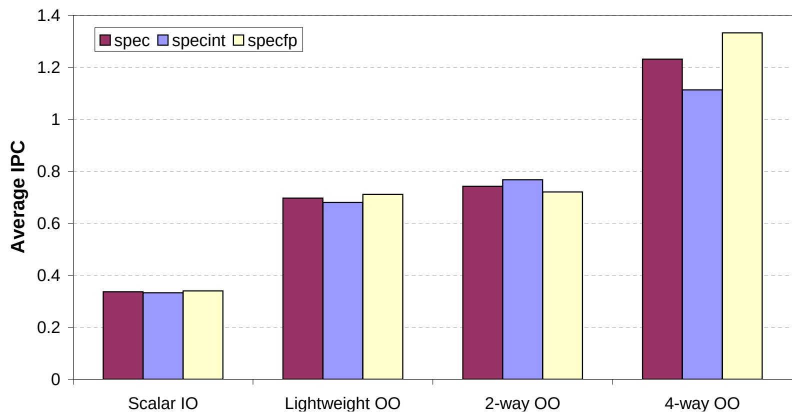

Figure 3.3: Arithmetic mean IPC to the lightweight scalar in-order core, the 2-way OOO core and traditional 2-way and 4-way OOO cores.

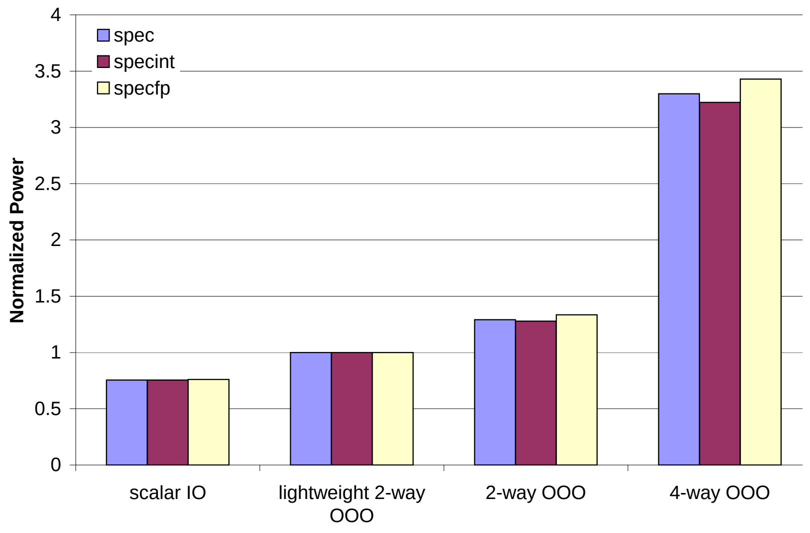

Figure 3.4: Arithmetic mean power consumption normalized to the 2-way OOO core for all four cores.

and 4-way OOO cores is shown in Figure 3.3. The lightweight core's performance is about 6.5% lower than the traditional 2-way OOO core's and 52% better than the scalar, in-order core. The 4-way OOO core has the highest performance as expected, outperforming the 2-way OOO core by 66%. Performance of the lightweight core on specint suffers relative to the traditional core due to the limitations of its simple branch predictor and commit-time branch recovery. On the other hand, its power usage of the lightweight core is 22.5% lower than the traditional OOO core's, as shown in Figure 3.4. This is primarily due to the much lower of the MAT and consumer-based issue queue compared to the CAM-based alternatives. The scalar cores has 24% lower power than the lightweight core, a much smaller difference than the difference in performance. The smaller difference is primarily due to leakage, which limits how much the lower activity factors of the scalar core can lower its overall power. The 4-way OOO core on the other hand uses 230% more power than the lightweight core thanks to the high power draw of its large structures.

Figure 3.5 shows the average energy efficiency of all cores in $\frac{BIPS^3}{Watt}$. [4]. The traditional

[4] $\frac{BIPS^3}{Watt}$ is like ED^2 in that both are voltage-independent metrics to capture the energy cost required for a

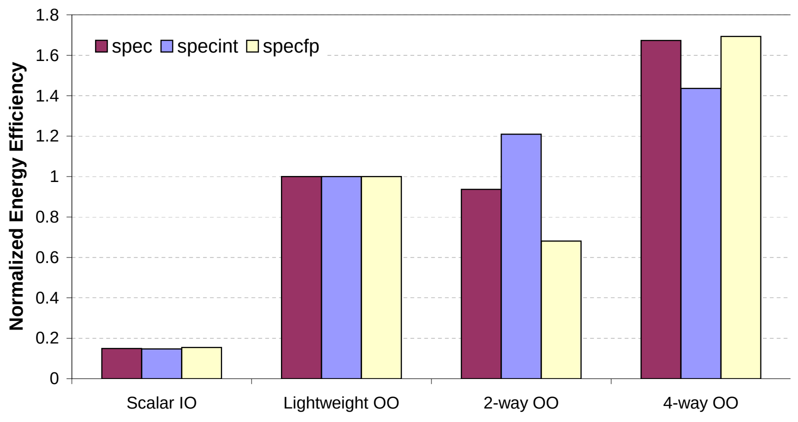

Figure 3.5: Arithmetic mean $\frac{BIPS^3}{Watt}$ normalized to the lightweight 2-way OOO core.

2-way core still beats the lightweight core on specint, but the lightweight core has a 6.7% energy-efficiency overall. The scalar core is very inefficient in this metric, dragged down by the combination of long execution time and high power floor due to leakage.

Because future manycores are will be limited by both power and area, I have developed a composite measure of power- and area-efficiency by dividing the power-efficiency number of each core (in $\frac{BIPS^3}{Watt}$) by the area of the core. Figure 3.6 shows this overall metric. The combination of smaller area, lower power and only modestly lower performance of the lightweight core results in it having a 36% higher combined power-and area-efficiency. Both the scalar core and the 4-way OOO core perform poorly in this metric, although for opposite reasons. The scalar cores has very low energy-efficiency combined with low area. The 4-way core has good energy-efficiency, but its area is almost 3 times as large as the lightweight core.

particular performance level. I prefer the BIPS-based metric because (unlike ED^2) larger values imply better results.

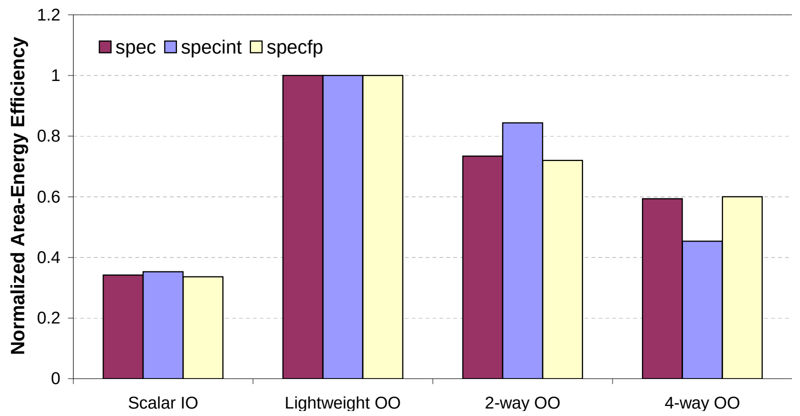

Figure 3.6: Arithmetic mean energy and area efficiency ($\frac{BIPS^3}{Watt \cdot mm^2}$) normalized to the lightweight 2-way OOO core.

3.7 Conclusion

Throughput-oriented cores in recent CMP designs have opted to forego out-of-order execution, judging it to not be power- and area-efficient. In this chapter I have shown that the largest structures needed for out-of-order execution can be redesigned to be more efficient, with only a 6.5% loss in overall performance. The lightweight core uses less overall power and more energy-efficient than a traditional 2-way OOO core. The lightweight core is also 22% smaller than the traditional OOO 2-way core and has better combined power- and area-efficiency than a scalar, in-order core, the traditional 2-way OOO core and a 4-way OOO core.

Chapter 4

Federation

Increasing difficulties in improving frequency and instruction-level parallelism have led to the advent of multicore processors. When designing such a processor, there is a fundamental tradeoff between the complexity or capability of each individual core and the total number of cores that can fit within a given area. For applications with sufficient parallelism, Davis *et al.* [32] and Carmean [25] show that maximum aggregate throughput is achieved by using a large number of highly multi-threaded scalar cores. However, for applications with more limited parallelism, performance would be improved with a smaller number of more complex cores.

How can these two approaches be reconciled? To improve the single-thread performance of an existing throughput-oriented system, one approach would be to add a dedicated out-of-order (OOO) core to the existing scalar cores. Unfortunately, this dedicated core comes at the cost of multiple scalar cores, reducing the aggregate throughput of the system. Using an even larger core with simultaneous multi-threading (SMT) would still limit the throughput and/or increase overall power. Instead, I propose *Federation*, a technique that allows us to retain a significant fraction of the performance benefit of the dedicated core with a much smaller area overhead.

4.0.1 Why Federation?

It may be objected that a better solution would be to have a small number of dedicated OOO cores to handle limited thread count. However, this approach cannot solve the problem for more than a few threads. This is because the area efficiency of OOO cores, even with SMT, is lower than that of multi-threaded in-order cores [32], which means that a CMP with a large number of OOO cores would have substantially lower throughput than a CMP using multi-threaded in-order cores. Certainly a dedicated OOO core will give great performance on *one* thread, and provisioning a single OOO core is a sensible solution to deal with the Amdahl's Law problem posed by serial portions of a parallel program or a single, interactive thread.[1]

I will show that Federation boosts performance with minimal area overhead, preserving the area efficiency advantage of multi-threaded in-order cores while offering performance competitive with dedicated OOO cores and the best energy-efficiency per unit area of all the options I studied.

Clearly, Federation can also be helpful for serial portions of execution if the designer chooses not to include a dedicated OOO core. This might occur if single-thread workloads or serial phases are not considered sufficiently important, or if design time or intellectual property issues preclude the use of a true OOO core.

4.0.2 Contributions

In this work, I first describe how to take two minimalist, scalar, in-order cores that have no branch prediction hardware and combine them to achieve two-wide, OOO issue. I also show how Federation, with some small adaptions, can be extended to dual-issue in-order cores, enabling the construction of a 4-way federated OOO core.

[1]You see this approach embodied in the Sony Cell [51] and AMD Fusion [48]

The main contributions of this chapter are:

- I show how to build a minimalist OOO processor from two in-order cores with less than 2KB of new hardware state and only 3.7% area increase over a pair of scalar in-order cores, using the lightweight structures introduced in Chapter 3. By comparison, a traditional 2-way OOO core costs 2.65 scalar cores in die area!
- I show that despite its limitations, such an OOO processor offers enough performance advantage over an in-order processor to make Federation a viable solution for effectively supporting a wide variety of applications. In fact, the two-way federated organization often approaches the performance of a traditional OOO organization of the same width, is competitive in energy efficiency with a traditional OOO core of the same width, and has better area- efficiency than all other options I studied.
- I show that it is possible to extend Federation to 2-way in-order cores and achieve performance close to a dedicated 4-way OOO core.

Federated cores are best suited for workloads which usually need high throughput but sometimes exhibit limited parallelism. Federation provides faster, more energy-efficient cores for the latter case without sacrificing area that would reduce thread capacity for the former case.

4.1 Background

Future microprocessor designs will likely incorporate many simple in-order cores rather than a small number of complex OOO cores [9]. Current examples of this trend include the Sun Niagara I and II [3, 56], each of which contain up to eight cores per processor. At the same time, graphics processors (GPUs), which traditionally consist of a large number of simple processing elements, have become increasingly programmable and general purpose [84]. The most recent GPU designs from NVIDIA [81] and AMD [7] incorporate 128 and 320 processing elements, respectively. This so-called *manycore* trend will provide substantial increases in throughput but may have a detrimental effect on single-thread latency. Federation is proposed to overcome this limitation.

When designing a federated processor, there are two possible approaches: design a new processor from the ground-up to support Federation or add Federation capability to an existing design. For the purposes of this chapter, I will take the latter approach and add Federation support to an existing multicore, in-order architecture. Based on the current trends cited above, the baseline in-order microarchitecture which I will focus on is similar to Niagara. It is composed of multiple simple scalar in-order cores implementing the Alpha ISA which are highly multi-threaded to achieve high throughput by exploiting thread-level parallelism (TLP) and memory-level parallelism (MLP) [44]. Specifically, each in-order core has four thread contexts[2], with hardware state for 32 64-bit integer registers and 32 64-bit floating point registers per thread context. Additionally, the integer and floating point register files are banked, with one bank per context and two read ports and one write port per bank. Unlike Niagara I (but like Niagara II), each core in our baseline architecture has dedicated floating point resources. To deal with multi-cycle instructions such as floating point instructions and loads, the in-order core has a small (four-entry) completion buffer. This buffer is used both to maintain precise exceptions and to prevent stalling when a multi-cycle instruction issues. The in-order cores implement only static not taken branch prediction and use a branch address calculator (BAC) in the decode stage to minimize fetch bubbles and to conserve ALU bandwidth.

To simplify the discussion, in this chapter I focus on a single pair of in-order cores

[2]But as Table 4.5 shows, the area overhead of multi-threading is not very large and Federation is thus an attractive option even for single-threaded cores.

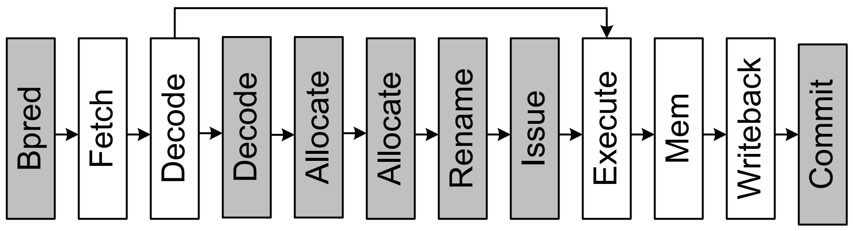

Figure 4.1: The pipeline of a federated core, with the new pipeline stages in shaded boxes.

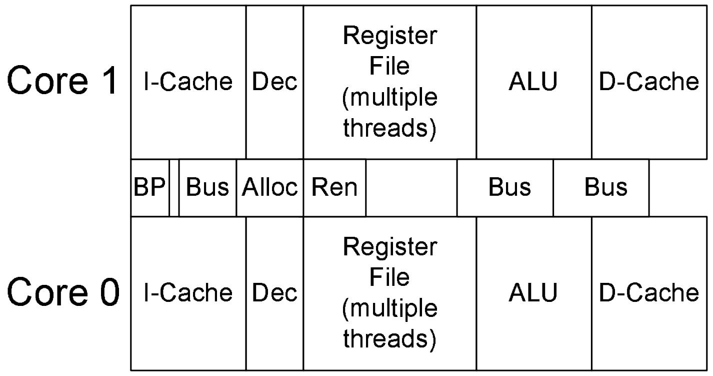

Figure 4.2: A simplified floorplan showing the arrangement of two in-order cores with the new structures necessary for Federation in the area between the cores.

which can federate to form a single OOO core. In practice, the techniques I describe are intended to be applied to a multicore processor with a significantly larger number of cores, with each adjacent pair of cores able to federate into a single OOO core.

4.2 Out-of-Order Pipeline

The primary goal of Federation is to add OOO execution capability to the existing in-order cores with as little area overhead as possible. Thus, each federated OOO core is relatively simple compared to current dedicated OOO implementations. Specifically, each federated core is single-threaded[3] and two-way issue with a 32-entry instruction window. The feder-

[3]Thus when the two in-order cores federate, the number of thread contexts provided by the pair of cores is reduced from eight to one. This clearly has implications for thread scheduling, which will be explored in

ated core implements the pipeline shown in Figure 4.1, with the additional pipeline stages not present in the baseline in-order cores shown in shaded boxes. A possible floorplan for the federated core is shown in Figure 4.2.

In order to limit the area overhead of Federation, I strive to avoid adding any significant CAMs or structures with a large number of read and write ports. Table 4.1 lists the sizes of the new structures required to support OOO execution, as well as whether or not each structure is implemented by re-using the existing hardware from the large, banked register file of the underlying multi- threaded core. While an extremely area-conscious approach could use the register file to implement all of the new structures, this would excessively increase the complexity and wiring overhead of the design. The structures which reuse the register file in our design are those which are close to the register read and writeback stages in the pipeline, require few read and write ports, and are read and written to at sizes close to those which the register file already supports.

The major new wiring required to support Federation is listed in Table 4.2. The following subsections provide a detailed explanation of the operation of each pipeline stage in the federated core, along with justification for the design tradeoffs that were made.

4.2.1 Branch Prediction

Federation uses the same minimal branch prediction and recovery as is described in Section 3.2. A new finite state machine (FSM) keeps track of which request to send to the instruction cache, deciding among misprediction recovery requests from the commit stage, the return address stack, and the NLS.

future work.

Structure	**Size (Bits)**	**Type**	**Reuses RF**
Branch Predictor (NLS)	6,144	SRAM	No
Branch Predictor (Bimodal)	4,096	SRAM	No
Return Address Stack	256	SRAM	No
Speculative Rename Table	640	Reg	No
Retirement Rename Table	384	Reg	No
Free Lists	384	Reg	No
Issue Queue (Wakeup)	176	Reg	No
Issue Queue (Data)	896	Reg	Yes
Unified Register File	4,096	Reg	Yes
Memory Alias Table	<64	Reg	No
Bpred Recovery State	256	Reg	No
Worst Case Total (Bits)	10,496 SRAM/6,844 Register		
Assumed Base Case Total (Bits)	10,496 SRAM/1,852 Register		

Table 4.1: Area estimates for the new structures added to the baseline in-order processor. Type differentiates between 6T SRAM cells as are used for caches and large tables and registers used for building the smaller structures inside the pipeline, which have full swing bitlines and are potentially multiported. The last column indicates whether I assume the structure can be built using only reused register file entries if the baseline core is multi-threaded. The worst case total is calculated under the assumption that none of the structure can reuse the register file.

New Wiring	**Width**
Cross Core Value Copying	2 * (64 + 6) bits
Mem Unit to 2nd D-Cache	2 * 64 bits
Cross I-Cache to Decode	32 bits
Decode to Allocate	approx. 64 bits

Table 4.2: The size of wires that must be added to the baseline core in order to support Federation.

4.2.2 Fetch

The fetch stage starts by receiving a predicted cache line from NLS, a return address from the RAS, or, in the case of a misprediction, a corrected PC from the branch unit in the execute stage. It then initiates the fetch by forwarding this information to the instruction cache (IC). The ICs of the two cores are combined into a cache with double the associativity and random replacement.

Since each core can only decode a single instruction, the second instruction (if valid) is sent to the second core for decoding. So that this extra wire does not influence cycle time, I allocate an extra pipeline stage (labeled "Decode" in Figure 4.1) for copying the instruction to the second core, buffering the first instruction in a pipeline register.

4.2.3 Decode

Once an instruction has been received from the fetch stage, the separate decode units in the two cores can operate independently. The decoded instructions are then routed to the allocate stage. If the first of the two instructions is a taken branch, a signal is sent to the allocate stage to ignore the second decoded instruction. Since the allocate unit is a new structure located between the two cores, propagating the instructions to it in the same pipeline stage as decode or allocate might influence overall cycle time. I instead allocate an extra pipeline stage (labeled "Allocate" in Figure 4.1) to allow the signals from both decode units to propagate to the allocate unit. The performance implications of this routing overhead are discussed in Section 4.4. The BAC of one of the baseline cores is used to calculate and verify the target of any taken branch.

4.2.4 Allocate

During the allocate stage, each instruction checks for space in several structures required for OOO execution. All instructions check for space in both the Issue Queue (IQ) and the Active List (AL). In traditional OOO architectures, load and store instructions would also need to check for a free Load-Store Queue (LSQ) entry, but our implementation uses a Memory Alias Table, which is free from such constraints (see Section 3.4). If space is not available in any of the required structures, the instruction (and subsequent instructions) will stall until space becomes available.

The allocate stage maintains two free lists, one for the IQ and one for the unified register file, with both lists implemented as new structures. I decided against using existing register file entries to implement these free lists because of their early position in the pipeline, the small size of each entry, and the complexity of deciding which entries to add to or remove from the free list. This complexity means that only a fraction of a clock cycle is available for the actual read/write operation. In addition to the free lists, the allocate stage also maintains the current AL head and tail pointers so that it can determine if there is space available in the AL and then assign an AL entry to the current instruction(s).

4.2.5 Rename

The federated core uses a unified register file with speculative and retirement Register Alias Tables (RAT). Since the design utilizes a subscription-based instruction queue (see Section 3.3), it must keep track of the number of subscribers for each instruction. For each architected register, its status and the number of consumers currently in the IQ is stored in a second table, which is accessed in parallel with the RAT.

Each rename table for a two-way OOO processor requires four read ports and two write ports, while each existing register bank has only two read ports and one write port.

Thus, implementing the rename tables using the existing register files would require the exclusive use of two entire register banks. Given the relatively small size of the rename table, it makes sense to implement it as a separate structure.

There are two separate OOO register files, one each for the integer and floating point registers. Each register file consists of the 32 architected registers and a number of rename registers, implemented using the register files of the underlying multi-threaded cores, with each register stored in both cores simultaneously. As mentioned earlier, the existing register files are heavily banked. The unified register files use part of several of these banks in order to support the required number of read and write ports. Even so, it is still possible for a particular register access pattern to require more reads from or writes to a single bank than that bank can support. Additional logic detects this case and causes one of the two instructions to stall. The performance impact of bank contention is explored in Section 4.4.

Logic is needed to check for read after write (RAW) dependencies between two instructions being renamed in the same cycle. Additional logic is also necessary to check for race conditions between an instruction being renamed and an instruction that generates one of its input operands being issued in the same cycle. This logic checks whether the status of one of the input operands is changing in the same cycle as its status is being read from the rename table. This classic two ships passing in the night problem is also present in many in-order processors, where instructions which check the poison bits of their input operands have to be made aware of any same-cycle changes to the status of those operands. Thus, depending on the design of the baseline in-order core, it might be possible to reuse this logic for the OOO processor. I assume that this capability is not supported by our baseline in-order core and that it must be introduced from scratch.

Because branches are only resolved at commit time, there is no need to checkpoint the state of the RAT for every branch. If a branch misprediction or another kind of exception is detected, the pipeline is flushed and a bit associated with each RAT entry is set to indicate that the most up to date version of the register is in the non-speculative RAT. As soon as an instruction in the rename stage writes to a particular register, this bit is reset to indicate that the speculative version is the most up-to-date.

4.2.6 Issue

Federation uses the consumer-based issue queue introduced in Chapter 3.

As mentioned in Section 3.3, the consumer-based issue queue does not issue older instruction first. In addition, schedulers for clustered architectures often attempt to schedule consuming instructions on the same cluster as their producers in order to avoid the overhead of copying the result between clusters. Given that our design maintains a copy of each register value on both cores, the core on which a consuming instruction is scheduled is only relevant in the case where it is ready to be issued as soon as its producer has issued. I again choose the simplest design, scheduling all instructions on core zero when possible and only assigning an instruction to core one when a previous instruction has been assigned to core zero that cycle. To avoid maintaining memory ordering across the two cores, loads and stores are only assigned to core zero.

4.2.7 Execute

Each instruction executes normally on the ALU to which it was assigned during the issue stage. The only change to the bypass network on each core is the addition of circuitry for copying the result to the register file of the other core. Since this is not a zero-cycle operation, the new circuits can be added without affecting the critical path. Additionally, a benefit of using the dependence-based IQ is that the core knows during execution whether it is necessary to broadcast the result using the bypass network, based on whether or not any consumers have subscribed to the instruction.

4.2.8 Memory Access

The data caches are merged in the same way as the instruction caches, by having each cache hold half the ways of a merged cache with twice the associativity. Instead of a traditional load-store queue, our design uses a simpler structure called a Memory Alias Table (MAT). I do not allow memory bypassing and flush the pipeline when a load and store are detected accessing the same address out-of-order. A detailed explanation and evaluation of the MAT is provided in Section 3.4. The only additional action required of load instructions in this stage is to index into the MAT with their target address and increment a counter.

4.2.9 Write Back

Similar to the Alpha 21264 [61], all results are written to the register files on both cores, to avoid the complication of having to generate explicit copy instructions for consumers on the other core.

4.2.10 Commit

Federation uses the commit time branch recovery that is described in Section 3.2.

4.3 Simulation Setup

I use the same simulator and inputs as describe in Section 3.5 for modeling the federated core as well as the cores I use for comparison.

The federated core is compared against five other cores: the baseline scalar, in-order core from which the federated core is built; a 2-way in-order core, designated federated in-order, built from two scalar cores; the lightweight 2-way OOO core; and traditional, dedicated 2-way and 4-way OOO cores. The simulation parameters for the different cores

Parameter	Scalar	2-way	4-way
Active List	none	32	128
Issue Queue	none	16	32
Load-Store Queue	none	16	64
Data Cache	8KB	16KB	32KB
Instruction Cache	16KB	32KB	32KB
Unified L2 Cache	256KB	256KB	2MB
Branch Target Buffer	none	512	4K
Direction Predictor	not-taken	2K bimodal	16K tour.
Memory	100 Cycles, 64-Bit		

Table 4.3: Simulator parameters for the different core types. The federated and lightweight cores have the same sized resources as the dedicated 2-way core. Note that the federated and lightweight cores use an MAT instead of an LSQ, and thus the number of loads and stores is limited by the size of the Active List rather than the size of the LSQ.

are listed in Table 4.3. Although the in-order cores are highly multi-threaded, the simulations run only a single thread, since this represents the best case for single-thread latency. Note that the smaller L2 for the small cores represents a single tile of a much larger L2, to simulate the fact that these cores will not be the only cores active on the chip and thus do not have exclusive use of the whole L2.[4]

4.4 Results

Figure 4.3 shows the impact on performance of the individual design changes of the federated core. Each energy saving or lower complexity feature is turned OFF individually to show its (negative) impact on overall performance; the IPC gain associated with each design choice thus represents the improvement in performance I would expect if the federated core instead used the associated more complex, traditional design approach. For example, the 1.74% improvement in IPC associated with the MAT indicates that I could improve the

[4]I also simulated all cores with a 32MB L2 cache and verified that while absolute performance improves by about 20%, this occurs across the board, so that the relative performance between the federated and the dedicated OOO cores changes by less than 0.9%

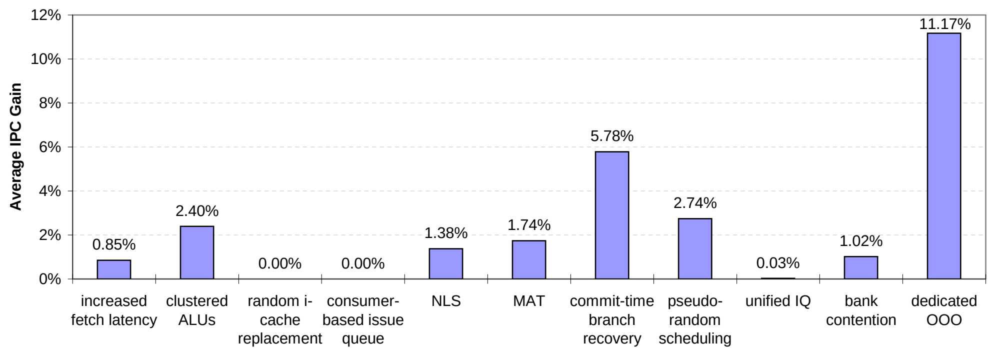

Figure 4.3: I show the performance impact of each individual feature by turning them OFF individually. The average IPC gain for a specific feature represents the performance improvement I would expect if I replaced that feature with the equivalent traditional, more complex design. The dedicated OOO data point shows the improvement in performance achieved by the dedicated OOO over the federated OOO core.

performance of the federated core by 1.74% by implementing an LSQ instead of a MAT. While most of the individual limitations have only a very small effect on performance, commit time branch recovery decreases average IPC by over 5%.

To separate out the impact of those features which I might apply to a traditional OOO core from the extra constraints imposed by federating two scalar cores, the two constraints which are a direct consequence of combining two distinct, baseline cores are shown on the left of the figure. These two constraints are the only constraints which do not apply to the lighweight core, which, as described in Chapter 3, is a dedicated 2-way OOO core with all of the low overhead structures of the federated core.

4.4.1 Other Points in the Design Space

The design chosen for the federated core represents only one point in a whole spectrum of possible designs. I have aimed for a balance between extra area and performance, but would also like to discuss some alternative design choices using the techniques I have presented which either provide greater area savings or increased performance. Commit time branch prediction recovery has a large negative performance impact on our design. The design tradeoff here would be to limit the number of unresolved branches in the AL at any given time and add a small number of shadow rename maps, which are saved on each branch and restored on a branch misprediction, to allow OOO branch recovery at write-back. Our experiments (not shown) reveal that adding only two shadow rename maps (768 register bits overhead) provides most of the benefit of OOO branch recovery and results in 5.1% better performance than the normal federated core. I did not use this configuration in the final analysis because I wanted to err on the side of the simplest design. Clearly this would slightly improve Federation's performance and energy efficiency.

The biggest additional structure of the federated core is the NLS branch predictor. To save even more space, I considered moving branch prediction from the fetch stage to the decode stage and only using a way predictor, reducing the number of bits in each NLS entry to the logarithm of the number of ways in the instruction cache. The target of direct branches would be calculated using the BAC, which is used to verify branch targets in all designs, and the NLS predictor would only predict which way of the set to read from the instruction cache. The most common indirect branches (returns) would be predicted by the RAS; however, the core would have to stall on other indirect branches. Using the way predictor would preserve the power savings associated with reading out only one way during most cycles, but reduce the size of the NLS from 6,144 bits to 1,536 bits. While the performance impact of moving branch prediction to the decode stage is only 0.5%, stalling on non-return indirect branches affects some programs significantly.

4.4.2 Area Impact of Federation

I used the same methodology as described in Section 3.5.1 to estimate the size of the federated core. The sizes of all cores used are shown in Table 4.5.

It is interesting to note that the ratio of the area of the 4-way OOO core to the area

Unit Name	**State in Bits**	**Size in mm^2**
Bpred Dir Table	$2 \cdot 2048$	0.0167
Bpred Target Table	$15 \cdot 512$	0.0313
Rename Tables	$4 \cdot 32 \cdot (5 + 4)$	0.0194
Consumer Inst Queue	00	0.0231
Inter Core Wires	NA	0.0515
Total	NA	0.1422

Table 4.4: Estimated sizes of extra structures for Federation in 45nm technology.

Core Type	**Size in mm^2**
1-way in-order	1.739
1-way in-order MT	1.914
Federated OOO	3.970
Lightweight 2-way OOO	3.945
2-way OOO	5.067
4-way OOO	11.189

Table 4.5: Estimated sizes for core types in 45nm technology.

of the in-order core is close to the 5-to-1 ratio in [25], even though our assumptions and baseline designs are somewhat different.

The area of the federated core was calculated by adding the areas of all the major new functional units to the area of two scalar in-order cores. I estimated the area needed by the major inter-core wiring listed in Table 4.2 by calculating the width of the widest new unit (the integer and floating point rename tables laid out side-by-side) and using the same 280nm wire pitch as used in [55]. In contrast to that work, which has a significant amount of extra area devoted to new inter-core wires, the area used by the wires for federating two cores is less than 0.05mm^2, since the wires do not have to cross over multiple large cores, but only connect two immediately adjacent small cores.

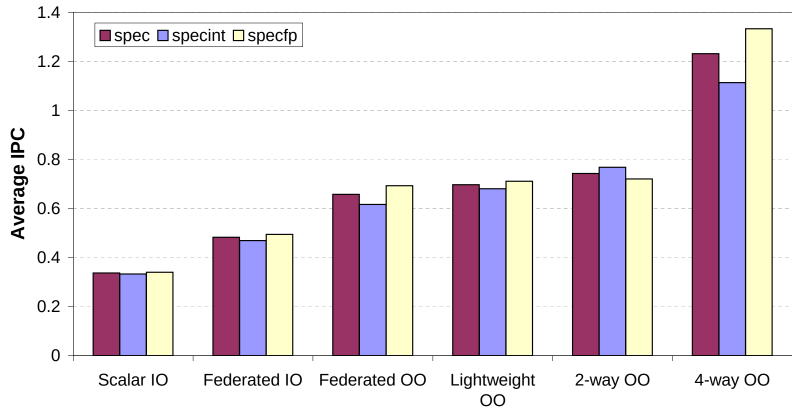

Figure 4.4: Arithmetic mean IPC.

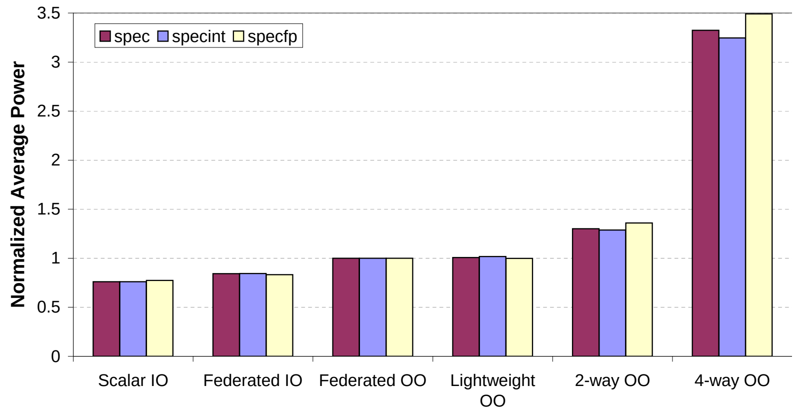

Figure 4.5: Arithmetic mean power dissipation, normalized to Federated OOO.

4.4.3 Overall Performance and Energy Efficiency Impact of Federation

The overall performance of the six different core types is shown in Figure 4.4, with their average power consumption shown in Figure 4.5. The 4-way OOO core achieves about twice the IPC of the federated OOO core but uses about three times the power, while the dedicated 2-way OOO core achieves 12.9% higher performance than the federated OOO core while dissipating 30.1% more power. The lightweight OOO core achieves 5.9% better performance than the federated OOO core with only a fraction of a percent higher power consumption. The dedicated in-order core and the federated in-order core have

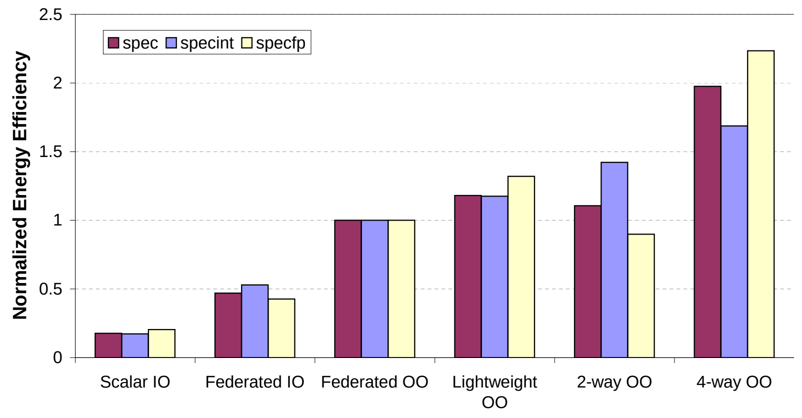

Figure 4.6: Arithmetic mean $\frac{BIPS^3}{Watt}$, normalized to Federated OOO.

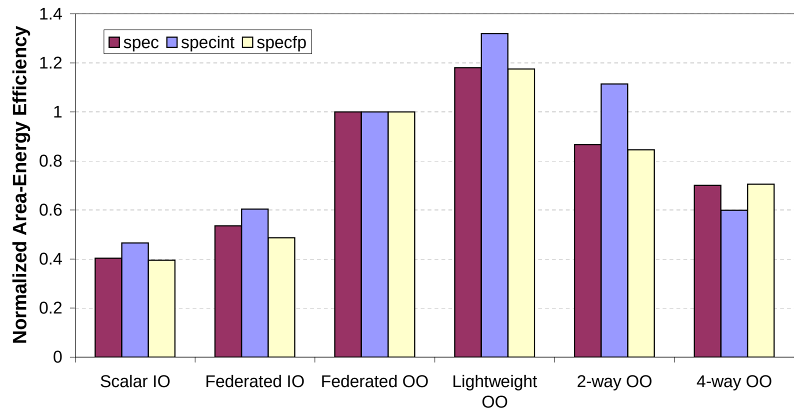

Figure 4.7: Arithmetic mean $\frac{BIPS^3}{Watt \cdot mm^2}$, normalized to Federated OOO.

substantially lower performance than the federated OOO core, which is not fully offset by their lower power consumption. This can be partially attributed to the fact that all cores—except for the 4-way OOO core, which has larger caches—have similar amounts of leakage in their caches and thus the savings in active power are offset to some degree by the static leakage power.

Figure 4.6 shows the average energy efficiency in $\frac{BIPS^3}{Watt}$ of the different cores.[5] The high-performance 4-way OOO core has a large advantage over the smaller cores in energy

[5] $\frac{BIPS^3}{Watt}$ is like ED^2 in that both are voltage-independent metrics to capture the energy cost required for a particular performance level. I prefer the BIPS-based metric because (unlike ED^2) larger values imply better results.

efficiency, because it is able to use its higher power to achieve substantially better performance. The dedicated 2-way OOO core has better efficiency than the federated OOO core in SpecInt, but lower efficiency in SpecFP. The lightweight OOO core has higher energy efficiency than the federated core thanks to its higher performance and essentially equivalent power dissipation. The two in-order cores have the lowest energy efficiency, even though they have the lowest absolute power consumption. Once again, this is mostly due to leakage power, which penalizes cores with longer execution times.

To measure both the power- and area-efficiency of the different cores, Figure 4.7 shows the $\frac{BIPS^3}{Watt \cdot mm^2}$ of the different configurations. The purpose of this metric is to account for the area cost of attaining a certain $\frac{BIPS^3}{Watt}$ value. In fact, this metric does not even show Federation's true benefits, since most of the area of the federated core is reused from the underlying scalar cores, whereas the area of the dedicated cores must be cannabalized from the existing cores. I am investigating a metric based on *extra* area required by a particular organization. Nevertheless, in terms of $\frac{BIPS^3}{Watt \cdot mm^2}$, the lightweight OOO core outperforms the federated OOO core by 18%, while the federated OOO core outperforms the dedicated, traditional 2-way OOO core by 13.3% and the 4-way core by 30%.

4.5 Federating 2-way Cores

In the previous section I explored federating two multi-threaded scalar cores into an OOO core, based on the assumption that scalar cores were the most efficient use of area for throughput. There have been several recent designs [56] which employ 2-way in-order cores, even when the power budget is very limited. Reasons for choosing 2-way cores instead of scalar cores might include an inability to include a single high-performance core along with the multiple throughput cores. Such systems need the higher single-thread performance a 2-way core can offer.

For designs which use 2-way in-order cores as their baseline, I explored federating two of these cores into a 4-way OOO core. While all the new structures I introduced for Federation can be scaled to support a 4-way core, I add some improvements to most structures to enable both higher performance and lower power when scaled.

4.5.1 Changes to Federation Structure

Many high performance OOO cores support predicting multiple branches per cycle. While the NLS can implicitly jump over non-taken branches, I do not extend either the direction predictor or the NLS predictor to produce multiple predictions per cycle.

Commit time branch recovery was already the biggest single performance cost in the 2-way federated core, and would have imposed a 15% performance penalty on the 4-way federated core (data not shown). Changing the processor to allow OOO branch recovery requires a small number of rename map checkpoints, as well as logic in the rename stage which steers updates of the rename map to the appropriate branch checkpoint. I found that four branch checkpoints delivered performance almost equivalent to having no limit to the number of branches in the ROB.

Simply scaling the dependence based issue queue to support 4-way issue would require doubling both the number of read and write ports as well as extending the arbitration logic to support issuing four instructions to the different ALUs. To reduce the number of ports required as well as the complexity of the arbitration logic, I use ideas from [118] to partition the issue queue among the issue ports in a fixed manner. For a federated core of 2-way baseline cores, the instruction queue is partitioned into four equal partitions. Each partition can only receive and issue a single instruction per cycle, but receives wakeup signals from all partitions. Instructions are assigned to issue queue partitions at rename time primarily based on which ALU type is assigned to which issue port, and secondarily on a load-balancing heuristic. As with load-balancing between cores and selecting among ready instructions, I choose the simplest mechanism possible of distributing instructions round-robin to partitions with empty slots.

Unlike issue queues in clustered architectures, which are distributed among the different clusters, assigning an instruction to a particular partition of the instruction queue does not mean a fixed assignment to a fixed ALU on a fixed core. For the case of a federated 4-way core, the partitioned instruction queue steers instructions to the two cores based on how many instructions are being issued in any given cycle. This is accomplished by taking the ready signals from the four partitions and feeding them into a four-entry priority encoder. The first two partitions with ready instructions get to execute their instructions on core zero, while the next two partitions execute their instructions on core one. For most benchmarks this steering policy means that the great majority of instructions are executed on core zero and do not incur any extra latency when sending or receiving values from the load/store unit. Because the ready information for instructions in the issue queue has to be available before select can occur, the inter-partition priority encoder can operate in parallel to instruction select and not impact the critical path.

An issue which parallels the problems of the issue queue is the increasing number of ports on the register file. While the number of read ports required by Federation is matched by the underlying cores, the number of write ports is not. To avoid having to increase the number of write ports, I use a technique similar to [65] of partitioning the unified register file between the different functional units. Using the banked register file of the underlying core, I assign one bank per issue port, reducing the number of write ports required to just one per bank.

In the initial implementation of the MAT, all loads and stores were treated as if they moved 64 bit values, the largest operand size in the Alpha ISA. Treating all loads and stores as uniform simplified the MAT implementation to only require a single counter

Parameter	**2-way IO**	**4-way OOO**
Active List	none	128
IQ	none	32
LSQ	none	64
Data Cache	32KB	64KB
Instr. Cache	32KB	64KB
Unified L2	2MB	2MB
BTB	512	4K
Dir Pred	2K bimodal	16K tour.
Memory	100 Cycles, 64-Bit	
Branch Misprediction Penalty	16 Cycles minimum	

Table 4.6: Simulator parameters for the 2-way in-order and 4-way out-of-order cores.

increment/decrement or check per operation, no matter what the actual operand size of the load or store was. The downside of dealing with all loads and stores in this manner is that extra aliasing will occur if adjoining 32 bit values are written to and read from in close proximity.

To eliminate this false aliasing problem, I changed the MAT implementation to support 32 bit loads and stores as default. Operations which move 64 bits must increment/decrement or check two adjoining counters in the MAT. This increases the complexity of the MAT's logic and makes the MAT appear half its size for 64 bit operations, but eliminates the problem of false aliasing between neighboring 32 bit values.

Because the 4-way core can still only issue one load and one store per cycle, the MAT retains the same number of ports as the base federated core. To support a larger number of memory instructions in flight without too many false positive memory aliasing events, I increase the number of entries in the MAT.

4.5.2 Simulation Setup

The simulation infrastructure described in Section 4.3 was also used for this set of experiments. The resources of the dedicated 2-way in-order and 4-way OOO cores are shown in

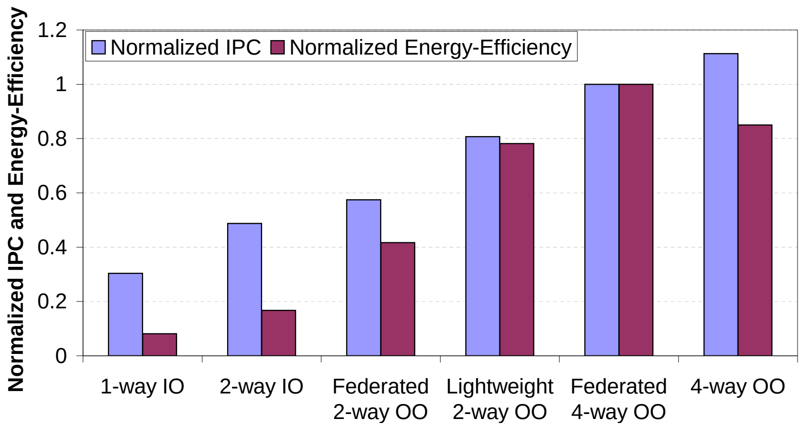

Figure 4.8: Arithmetic mean IPC and $\frac{BIPS^3}{Watt}$, normalized to Federated 4-way OOO.

Table 4.6. To reflect the greater emphasis on single-thread performance that a design using 2-way in-order cores might have, I substantially increased the pipeline depth of all of the core types to more accurately represent designs which aim at achieving higher frequencies.

I compare the 4-way federated core against five other cores: the scalar core used as the baseline for the 2-way federated core; the 2-way in-order core used as the baseline for the 4-way federated core; the 2-way federated OOO core; the lightweight 2-way OOO core; and the dedicated 4-way OOO core. Here, the resources of the lightweight core have been scaled to match those of the the dedicated 4-way OOO core.

4.5.3 Results

Figure 4.8 shows the relative performance and energy efficiency of the six core types. The 4-way federated core achieves performance only 10% worse than the dedicated 4-way OOO core. Comparing the $\frac{BIPS^3}{Watt}$ of the different cores shows that the 4-way federated core provides 15% better energy efficiency than the dedicated 4-way core. This result shows that even large OOO cores can benefit heavily from more power efficient structures, as long as they do not impact performance too significantly.

The changes to the Federation structures outlined in Section 4.5.1 impact performance as follows: the improved and enlarged MAT boosts performance by 3% due to fewer false positive memory aliasing events; the partitioned instruction queue surprisingly does not hurt performance on average; and, as previously mentioned, the introduction of branch checkpoints improves performance by 15% and is the single largest contributor to the improved performance of the 4-way federated core.

4.6 Conclusions and Future Work

Manycore chips of dozens or more simple but multi-threaded cores will need the ability to cope with limited thread count by boosting the per-thread performance. This chapter shows how 2-way OOO capability can be built from very simple, in-order cores, with performance 92.4% better than the in-order core, 30% lower average power than a dedicated 2-way OOO core, and competitive energy efficiency compared to a 2-way OOO core. Using a consumer-subscription based issue queue and eliminating the Load-Store Queue in favor of the Memory Alias Table, I have shown that no major CAM-based structures are needed to make an OOO pipeline work. In fact, these same insights can be used to design a new, more efficient, OOO core, as the lightweight OOO results show. However, even a lightweight dedicated OOO core would still come at a high cost in area. I have also shown that the techniques of Federation can be applied to higher performance 2-way in-order cores to achieve performance close to that of a dedicated high-performance 4-way OOO core.

The most important advantage of Federation is that it can be added to a manycore architecture without sacrificing the ability to use the constituent in-order cores as multi-threaded, throughput-oriented cores. Federation requires several new structures, but with very low area overhead—less than 2KB of new SRAM tables and less than 0.25KB of new register-type structures in the pipeline per *pair* of cores—only 3.7% area overhead per pair.

Put another way, this means that for a set of 32 scalar cores, the area of Federation for each pair only adds an aggregate area equivalent to 0.59 cores or 0.373 MB of L2 cache. For 2-way in-order cores with branch prediction the relative area overhead is even less. As a result, Federation actually provides greater energy efficiency per unit area—specifically, 13.3% better $\frac{BIPS^3}{Watt \cdot mm^2}$ than a dedicated 2-way OOO core, and 30% better than a 4-way OOO core!

The option of adding Federation therefore removes the need to choose between high throughput with many small cores or high single-thread performance with aggressive OOO cores and the associated problems of selecting a fixed partitioning among some combination of these. This is particularly helpful in the presence of limited parallelism and it allows a multicore chip to trade off throughput for latency on a very fine-grained level at runtime. Federation thus allows multicore chips to give higher performance across a wider spectrum of workloads with different amounts of TLP and deal with workloads that have different amounts of parallelism during different phases of execution.

As I have pointed out in Section 2.2.1, the structure of Federation was chosen with the lessons of clustering in mind. As such, I designed Federation without further plans for horizontally aggregating more than two cores into a single very wide core. For higher single-thread performance, the combination of Federation with techniques which can effectively shorten the critical path — such as runahead execution [78], sophisticated prefetchers [41], or dynamic optimization [4] — seems to be the most fruitful path to pursue. Many such techniques have as one of their main advantages their toleration of infrequent or long latency communication with the main core, which makes it much easier to implement them using multiple cores of a manycore processor. Future work on using manycore processors to improve single-thread performance will have to find the right balance between adding extra hardware when absolutely necessary and emulating many hardware features with software or firmware on some of the cores of the processor.

Chapter 5

Diverge on Miss

5.1 Introduction

The growth in single-thread performance has slowed dramatically in recent years, due to limits in the power consumption, thermal hotspots and complexity of microprocessors. As a response, the microprocessor industry has shifted its focus onto multicore processors, which combine a number of cores onto a single die. Some of these designs give higher priority to overall throughput than to single-thread latency, trading out-of-order cores for simpler, smaller in-order cores which are smaller and less power hungry. While single-thread performance suffers, overall chip throughput is increased. This design point is often referred to as manycore, as opposed to more traditional multicore designs, which retain large, high-performance out-of-order cores for maximum single-thread performance.

In the previous Chapters I assumed that multithreaded, scalar in-order cores would be the throughput cores of future asymmetric manycore processors. *Single instruction multiple data* (SIMD) cores offer an attractive addition to scalar cores, because SIMD organization can amortize the area and power overhead of a single frontend over a large number of execution backends. For example, using the same area estimation methodology I use in Section 3.5.1, I estimate that a 32-wide SIMD core requires about one fifth the area of 32 individual scalar cores. [1]

If SIMD cores have better throughput, power- and area-efficiency than scalar cores, the design point for cores between the throughput cores and the few, high-performance cores will probably move. One possibility is that slightly fewer cores with higher performance would be preferred compared to scalar, in-order cores, making the lightweight out-of-order cores cores or 2-way in-order cores which can be federated an optimal design choice.

For SIMD cores to be accepted as the main throughput core type of asymmetric CMPs, their performance must be consistently better than scalar cores across the widest possible range of programs. This is currently often not the case due to limitations in how SIMD cores perform on certain memory access patterns.

5.1.1 Divergent Memory Accesses

To better tolerate memory and pipeline latencies, SIMD manycore processors typically use fine-grained multithreading, switching among multiple warps[2], so that active warps can mask stalls in other warps waiting on long-latency events. The drawback of this approach is that the size of the register file increases along with the number of warps per core. Most current and planned manycore processors also use on-chip caches to reduce the required off-chip bandwidth and to hide the latency of accessing DRAM as much as possible. The combination of SIMD cores and caches presents special problems for architects because each SIMD thread may independently hit or miss. This problem is not just limited to *array-style* SIMD organizations where each SIMD thread is a scalar processing element. *Vector-SIMD* instructions sets with gather support, including [97, 1] suffer the same problem. Divergence becomes a particular problem for load or store instructions that have irregular

[1]Note that this estimate does not include the area of any interconnection network, among the MIMD cores, which often grows supra-linearly with the number of cores [69].

[2]For simplicity, I use the term *thread* to refer to a SIMD lane, and *warp* to a SIMD group that operates in lockstep. Multithreading a SIMD core therefore consists of supporting multiple warps.

access patterns. Consider code where each thread of a SIMD warp needs to read many contiguous values in a global array, but each thread accesses distinct regions, starting at a random offset, for example in DNA sequence alignment. While reading in their values, the probability that a thread in a warp will cross a cache line boundary and have to stall grows as the number of threads per warp increases. In such a case the lockstep nature of SIMD execution forces the core to stall or switch to another warp for each load. Clearly, such memory access patterns will waste much of the computational power of the SIMD core waiting on memory requests.

Here I present a new hardware mechanism, *diverge on miss*, that takes advantage of looping behavior to temporarily mask off threads in a warp that miss in the data cache and allows the other threads to continue executing, re-enabling the masked off threads as soon as possible. Letting threads which hit in the cache continue to execute allows them to use idle execution slots when all warps of a core would otherwise be stalled. It also allows them to issue future cache misses earlier, increasing memory level parallelism [44].

We show that diverge on miss can increase performance of a manycore processor using 32-wide SIMD cores by up to a factor of 3.14, can decrease the area of each SIMD core by 35% at equal performance or increase peak performance by 30%. We show how such a mechanism can be built with low-overhead on top of existing structures meant to deal with control-flow divergence. Diverge on miss builds on the fact that high-performance SIMD and vector cores *already have* logic for masking off threads on a fine-grained basis to support arbitrary control-flow and *can already deal with* multiple parallel memory operations finishing out-of-order due to their support of scatter/gather operations.

5.2 Background on SIMD Divergence Handling

5.2.1 Control-Flow Divergence

The baseline architecture in this study uses the same post-dominator based reconvergence algorithm as presented in Fung *et al.* [39]. Each warp is associated with a branch divergence stack, which tracks control flow for all threads in the warp. Each entry in this stack holds 3 fields, the active PC field, an active threads bitmask and a reconvergence PC field.

If a divergent branch (where some threads evaluate the branch as taken and some as not-taken) is executed, the top of the stack entry is modified to hold the reconvergence PC along with a bitmask of the currently active threads in the warp. A new entry is pushed on the stack consisting of the fall through PC, a bitmask indicating which threads evaluated the branch as not-taken, as well as the reconvergence PC of the branch. A second entry consisting of the branch target PC is also pushed on the stack, along with the bitmask indicating which threads evaluated the branch as taken, and again the reconvergence PC.

The active PC and thread active bitmask are then set to the active PC and bitmask fields of the top of the stack (which is the taken branch entry in this case) and execution continues. When the active PC reaches the reconvergence PC, the stack is popped and the active PC and bitmask are set to the values contained in the not-taken stack entry.

Finally, when the reconvergence PC is reached a second time the active bitmask is restored to what it was before the branch. If a branch is encountered multiple times in a row (such as a loop branch), then no new entry needs to be created on the stack; it is enough to modify the bitmask if any active threads want to exit the loop. As we will show in Section 5.3, the same basic operations that are needed to support control-flow divergence by the pipeline logic (checking the PC against a PC stored in a structure, taking a pre-defined action if the PC's match, modifying the bitmask of active threads based on the result of that action) also to support diverge on miss.

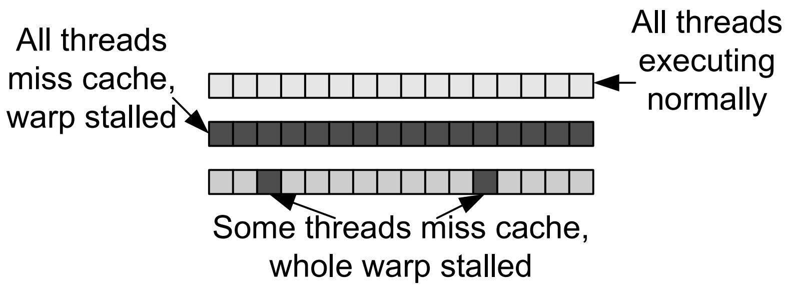
All threads miss cache, warp stalled
All threads executing normally
Some threads miss cache, whole warp stalled

Figure 5.1: Warps can be forced to wait on memory by a single miss from a single thread. Even cores with multiple warps are forced to stall by a single cache miss per warp.

5.2.2 Handling of arbitrary scatter/gather memory requests in the base architecture

Consider a SIMD vector or array core with scatter/gather support and an attached data cache. When such a core executes a load, the data cache looks up each cache line touched by each load from each thread. If even a single lookup misses, execution of the entire warp has to stall until that miss has been serviced. We call memory operations in which some threads hit and some threads miss *divergent* memory operations.

If a core only has a single warp to execute, it has to stall in such an event. Even a core with multiple warps that it can switch among can be stalled by only a small number of individual memory requests missing the cache, as illustrated in Figure 5.1.

If the architecture allows writing back individual thread register values into the SIMD register file as a background operation, no intermediate storage is needed. If this is not the case, a Memory Coalescing Buffer (MCB) is needed, where values are buffered between the time they are read from the cache and when they are written back. An MCB is also needed for divergent memory operations. All threads that have hit in the cache must capture their values, as the cache lines they access may be evicted during the servicing of any misses.

5.3 Diverge on Miss

Diverge on miss is a hardware mechanism which allows some threads in a warp to continue to execute on divergent memory accesses. Threads which miss in the data cache (or a given cache level if there is a multi-level cache hierarchy) are masked off and do not continue execution, while the threads that hit in the cache continue to execute normally. Such a warp is called a *slipping* warp, as it allows some threads to slip or lag behind others. Memory requests from missing threads are serviced in parallel with the warp continuing execution. When the warp next encounters the same memory instruction (or a condition which forces re-synchronization of all threads in a warp) the missing threads that have subsequently received their memory value are re-enabled if they have received their memory values. Threads which still have not received their memory values continue to be masked off. Individual threads can slip a variable amount relative to other threads, potentially missing the cache shortly after being re-enabled. Slipping warps can either catch up when other threads miss in the cache or continue to execute after the other threads have already finished executing, forcing the warp to execute longer.

For programs which are memory latency bound, diverge on miss can dynamically trade execution cycles for more latency tolerance, higher MLP and potentially improved utilization of the data cache. We will show in Section 5.4 how the hardware can use runtime control mechanisms to limit the amount of slip, controlling the amount of extra execution cycles based on the needs of the running program.

We discuss two options for supporting diverge on miss: a pure hardware implementation and a hybrid hardware-software approach which only exposes a new type of load and store instructions, but leaves all the implementation and handling of the divergence to the software layer.

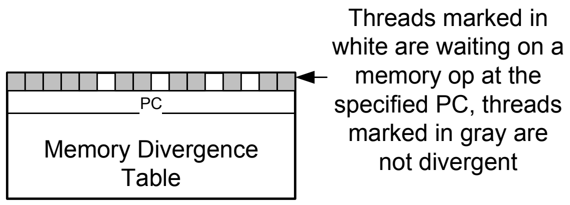
PC
Memory Divergence Table
Threads marked in white are waiting on a memory op at the specified PC, threads marked in gray are not divergent

Figure 5.2: The Memory Divergence Table tracks which lanes of a warp are waiting on which memory op and which ones are ready to be merged back into the active warp.

5.3.1 Pure Hardware Implementation

Diverge on miss uses a very similar structure to the divergence stack used by branch divergence. The Memory Divergence Table (MDT) shown in Figure 5.2, keeps track of divergent memory operations.

The following actions occur when a divergent memory operation is executed:

1. The memory request is issued to the cache and a bitmask indicating which threads hit and which miss is returned.

2. The fact that some threads hit and some missed is detected by the control logic.

3. The control logic searches the current warp's MDT entries for an existing entry with the same PC, merging the new request into the MDT entry if it exists. If an MDT entry is not found, the control tries to allocate an MDT entry to the instruction. Allocation might fail because of a limited number of entries per core, or because the adaptive slip controller (described later) decides that it is better to have this memory operation execute as a normal load or store [3].

   If allocation succeeds, the threads which missed the cache are written to the MDT as a bitmask along with the PC of the memory operation. The MCB entry is also

[3]In either case an MCB entry is also allocated to the memory operation. If no MCB entry is available, execution has to stall until an entry becomes available.

initialized with the memory addresses requested by the threads that missed the cache and the per-thread status fields are set to either waiting or invalid. The missing addresses are sent to the memory subsystem to be fetched while threads which hit in the cache receive their memory values and continue executing. If allocation fails the warp falls-back to normal SIMD execution, and blocks waiting for all misses to complete.

4. As the time between when a cache line is returned and when a thread can be merged back into the active warp cannot be known a priori, it is possible that a cache line would be evicted while the requesting thread is waiting for re-activation. To prevent this case, as soon as a cache line is returned the memory values that were requested are extracted and each value stored in the appropriate slot in the MCB entry. The slots' status bits are also updated from waiting to ready.

5. When the same memory instruction gets executed again (or a forced reconvergence happens), the control logic will again search the MDT and find an existing entry. Threads which have their status bits set to ready will write their value back to the register file along with those lanes that hit in the cache, and their status field will be updated to invalid. If all threads have the invalid status the entry can be deallocated.

5.3.2 Software-Controlled Implementation

An alternative approach is to add a new type of instruction, called the *load&snoop* and *store&snoop*. These instructions operate as normal loads and stores if they hit in the level one data cache (or another level of the cache hierarchy). If they miss, however, they do not block but are instead turned into implicit prefetches. By guaranteeing a fixed latency to completion they have the benefit of being easy to schedule for the compiler in optimized loops, similar to accesses to scratchpad memories in other architectures [38].

If a thread misses in the cache, a bit is set in a bitmask. The bitmask can be stored either in a special purpose register or returned as a second register write of the instruction, similar to the low and high parts of a multiplication. In this approach neither the MDT nor the MCB are implemented in hardware. The software can implement most of the functionality of these structures, or modify them according to the needs of the application. Note that because the cache lines which are prefetched are not locked down in any way, a *load&snoop* or *store&snoop* can fail repeatedly and indeed indefinitely. For example if all threads in a warp try to load distinct cache lines that are mapped to a single set in the cache and the cache's associativity is smaller than the width of a warp, it is impossible for all loads to hit in the cache simultaneously. It should be noted that software can always serialize all loads or stores of a warp if it detects too many retries.

5.3.3 Ensuring Reconvergence

Supporting SIMD divergence on memory operations raises similar concerns as supporting SIMD branch divergence. Ensuring that all threads get re-merged into the active warp and finish executing requires some extra policies and logic per core.

In typical usage a divergent load or store will be inside a loop body and executed a large number of times. In this scenario diverged lanes can normally reconverge on the next iteration of the loop. But if a thread diverges during the last loop iteration or control flow jumps outside the loop body must be ensured to still reconverge.

If a subset of threads in a warp reach a return statement while other threads are still masked off, the control logic checks the MDT and re-activates those threads while masking off the threads which have hit the return statement. Note that this is the same mechanism that is used to handle branch divergence, so the control logic only has to be extended to check the MDT in addition to the branch divergence stack. If there are multiple entries in the MDT this process is repeated until the MDT is empty.

5.4 Limiting Thread Divergence

A SIMD core which allows threads to diverge on cache misses has to deal with the problem of excessive divergence. This can happen if some threads hit in the cache the great majority of the time, while the others almost always miss. This can happen due to the inherent nature of a given workload, the interaction of the program with the cache subsystem or a number of other reasons. In the worst case this means that by the time the fast threads finish executing a loop, the slow threads have only advanced a few iterations. The warp containing these threads will have to execute the slow threads to completion, greatly wasting execution cycles and not gaining any benefit in terms of overall warp execution latency.

Worse, excessive divergence is that it can make the cache access behavior of a given warp much worse, with accesses that would have been a contiguous, coalesced set of hits turning into accesses spread over multiple cache lines, increasing cache churn and decreasing hit rates. These drawbacks to diverge on miss SIMD execution grow proportionally to the divergence between threads in a warp.

To limit the amount of divergence I introduce new control hardware, which I call the Adaptive Slip Controller (ASC), to limit how far threads in a warp can slip relative to each other. The ASC has a small counter for each thread in a warp. If an undiverged warp encounters a diverge on miss event, those threads which hit in the cache have their counters incremented. If any thread's counter hits some maximum value, the warp reverts to blocking execution of all loads and stores until the maximum counter value falls below the maximum value again. Note that threads which are marked as inactive by the branch divergence stack are not considered in this process.

If a warp is already diverged when it encounters another diverge event and all tail-end threads (which have counter values of zero) hit in the cache, the counter values of all threads which miss the cache in this instance are decremented. The same mechanism applies when some threads reach the maximum counter value. They are disabled and their counters get decremented when the remaining threads hit in the cache. The counter of each thread is reset when hardware warps are reassigned to a new set of software threads.

5.4.1 Adaptively Limiting Thread Divergence

The optimal maximum divergence value is very much dependent on the interaction of the program, the input and the architecture. We use a mechanism - called adaptive diff - which keeps track of the number of cycles a core has been was not actively executing instructions (a value of zero indicating that it is completely ALU bound), if the amount of off-chip bandwidth that it used was above its fair fraction of overall bandwidth (bandwidth bound), and the number of cycles it was stalled waiting on memory (latency bound). Since enabling more slip results in more extra execution cycles (as the trailing threads finish execution) and can result in more bandwidth usage (due to previously coalesced accesses being broken into chunks which are touched at different points in time), the amount of slip is controlled by how ALU, bandwidth or latency bound a given program is during a sampling period. I use very long sampling periods of 100000 cycles or more. If the core was neither ALU nor bandwidth bound over a given sampling period, the maximum allowed divergence value is incremented, and otherwise it is decremented.

5.5 Hardware Overhead

Diverge on miss adds the Memory Divergence Table, the per-thread divergence counters and some other small structure to each core. Table 5.1 lists the extra state required for each

Structure	Fields per Entry	State per Entry	Number of Entries	Total Structure Size
Memory Divergence Table	PC, Thread Bitmask	32 bits +32 bits	2·N	16 – 256 bytes
Per-Thread Divergence Counter	Per-Thread Counter	8 bits	32·N	32 – 512 bytes
Per-Warp Slip-Limit Bitmask	Thread Bitmask	32 bits	N	4 – 16 bytes
Max-Slip Counter	Per-Core Counter	8 bits	1	1 byte

Table 5.1: New structures needed to support diverge on miss. N is the number of warps per core, which range from 1 to 16.

structure. The MDT is similar to the branch divergence stack, in that each entry needs to record the PC of a divergent instruction, along with a bitmask indicating which threads took which of the two possible paths. The number of MDT entries per warp is directly related to the maximum number of outstanding memory operations each warp supports. I assume that the baseline architectures allows two outstanding memory operations per warp, which means that the augmented core with diverge on miss has two MDT and MCB entries per warp.

As explained in Section 5.4, it is useful to dynamically adapt the maximum amount of slip allowed between threads in a single warp at runtime. To track the slip of each thread a small counter is needed per thread. I assume that each counter is 8 bits, allowing threads to slip by 255 hits or misses relative to each other. These counters are updated with each divergent memory operation and checked against the Max Slip Counter. If any thread reaches the maximum allowed slip a bit is set in the warp's Slip-Limit Bitmask, disabling further execution of that thread until divergence is reduced below the threshold value.

core type	**Area**
scalar core	1.05 mm^2
32 scalar core	33.60 mm^2
32-wide SIMD core with 2 warps	7.3 mm^2
32-wide SIMD core with 16 warps	11.5 mm^2

Table 5.2: Area estimates for different core configurations

5.5.1 Core Areas

To estimate the area of the SIMD cores, I used the same methodology as used in Section 3.5.1. I assume that each lane in a SIMD core has a 32 bit data path and that each thread has a total of 32 32-bit registers, so that each 32-wide SIMD warp uses 4KB of register file.

I use the numbers for each functional unit and scale them by their capacities and port numbers relative to the Opteron core. Table 5.2 shows the areas for a 32-wide SIMD core with 2 warps, a core with 16 warps, a scalar core and 32 scalar cores calculated with this methodology.

5.6 Experimental Setup

5.6.1 Simulator

My custom simulator models a number of SIMD/vector cores, along with a cache hierarchy and a shared memory subsystem. The cores are modeled as having a constant CPI of one for all non-memory instructions and having private L1 data caches and that the structures for holding outstanding memory requests are not a limiting factor. Each core can have one or multiple warps, and it can switch among on a cycle by cycle basis at no extra cost. The scheduling algorithm is round-robin, skipping warps which are waiting on memory requests. The memory reference traces are collected directly from the native applications, which are instrumented with calls to my simulator. To determine the number of instructions between memory references, each application is inspected manually and the number of arithmetic and control flow instructions between memory references are passed to the simulator.

Direct instrumentation of native applications was preferred over gathering large memory traces to avoid the I/O and decompression overheads of normal trace based simulators. The combination of a simple core model and direct instrumentation of native applications allows the simulator to be very fast (slowdowns only about 10x over pure native execution are the norm) and can consequently capture the performance on input sizes which would be prohibitively slow to simulate otherwise. This is especially important when dealing with a large number of cores and threads per core.

5.6.2 Simulated System and Power Model

The base chip consists of 32 in-order cores each supporting 32-wide SIMD execution, all running at 2 GHz, for an overall maximum execution bandwidth of 2 Teraops. Each core has a 32KB private data cache, which has 32B cache lines and is 4-way set associative. I model a standard LRU replacement policy. All cores share a 256 GB/sec memory interface, with a memory access latency of 500 cycles.

5.6.3 Workload

The chosen application kernels represent a mix of application domains and memory access patterns. We have included a kernel (k-means) which is pure streaming, having no reuse of data between threads and cores. We do not expect this kernel to benefit from the sharing tracker, and use it to make sure the sharing tracker does not hurt such applications. Another set of kernels (neighbor list generation, Lennard-Jones force calculation and Gaussian filter) has data reuse between software threads, but the sharing patterns are mostly between threads that tend to access nearby data. These threads are often mapped to the same core and the L1 data caches are enough to capture most of the data reuse. We expect these kernels to show only limited benefits from the sharing tracker, as only a small fraction of memory requests will not hit in the local cache or go to global memory.

Lastly, we have also included kernels (ray tracing and DNA sequence alignment) which have both large working sets and data sharing patterns that are non-regular, meaning that threads on different cores will share data. We expect these kernels to show the most improvement out of all kernels.

5.6.4 Molecular Dynamics

We use the molecular dynamics package HOOMD (Highly Optimized Object Oriented Molecular Dynamics) [8] version 0.8. HOOMD is a general purpose molecular dynamics package that can take advantage of the computational power of GPUs using CUDA [80]. The two most computationally intensive functions in HOOMD are the Lennard-Jones potential computation and neighbor list generation, making up over 95% of the runtime. Note that HOOMD also supports other potentials, which all have the same computation and memory patterns as the Lennard-Jones computation.

The neighbor list function (NL) determines for every particle being simulated which other particles are close enough that their Lennard-Jones interactions with the current particle have to be taken into account. Since all particles move during the simulation time frame, the neighbor list is regenerated every 10 time steps. To avoid the need to check every particle against every other particle, particles are sorted into spacial bins in a preliminary step. Each particle then computes the distance between it and all of the particles in all the neighboring bins, adding those particles that fall inside of a cutoff radius to its neighbor list. To avoid having to regenerate the neighbor list each time step, the cutoff radius is made larger than necessary, so that particles which might move inside the real cutoff radius in several time steps are also added to the neighbor list.

The Lennard-Jones function (LJ) calculates the Lennard-Jones potential for each particle each time step, calculating distance and force for each particle on the neighbor list. Both kernels are parallelized by assigning each particle to a single thread.

I run the standard HOOMD benchmark simulating a liquid consisting of 64000 particles at a packing fraction of 0.2 interacting via the Lennard-Jones force. I simulate the first 600 time steps.

5.6.5 DNA Sequence Alignment

We use the program MummerGPU [96] (SA), which uses a suffix tree to efficiently find alignments of short DNA sequences (such as those generated by high-speed DNA sequencing machines) against a reference genome. The tree is traversed from the root in a data dependent manner, with each edge holding a variable number of base pairs which must all match for the traversal to proceed to the next node.

MummerGPU parallelizes its computation by mapping each input string to a thread. Similar to Schatz *et al.* [96], I run SA in the exact matching mode, matching batches of synthetic snippets of length 25, 50, 200 and 800 base pairs sampled randomly from the *Bacillus anthracis* genome $(GenBankID : NC\_003997.3)$ to match against itself. Each batch contains a total of one million base pairs, with batches containing longer string containing linearly fewer samples. I report the average performance over all 4 string lengths.

5.6.6 Ray Tracing

We use the bwfirt ray tracing framework [88], and specifically the provided SimpleBVH ray tracer as the test application. SimpleBVH decomposes the scene into a bounding volume hierarchy tree. Each ray traverses the tree to find the object that it hits in the scene. Bwfirt uses SimpleBVH to do path tracing through a given scene, letting rays bounce around a scene multiple times until they hit a light source. We chose bwfirt because it doesn't just trace primary rays, but use ray tracing to create effects which are very expensive to replicate with traditional GPU rasterization and can increase the quality of rendered images.

We parallelize SimpleBVH by having each thread trace a different ray through the scene. This method of parallelization provides a large number of independent tasks without the need for any communication between threads until the output of the final result. As input we use the conference scene with approximately 1 million triangles and set the resolution of the generated image to 1024 by 1024 pixels.

5.6.7 Data Mining

We use the k-means program (KM) from Minebench [79]. The k-means code randomly generates N cluster centers, where N is given by the user. It then computes the distance between each point and each cluster center and assigns each point to the cluster with the closest center. After completing the reassignment of points to clusters it recomputes the cluster centers as the average of all points assigned to the cluster. The last two steps are repeated until the number of points switching cluster to another falls below a pre-specified threshold.

Both the distance computation per point and the recomputation of the cluster centers can be easily parallelized. We assign each point to a thread for the distance computation as well as the cluster center recomputation. We run k-means with 32 clusters and with the provided input set of roughly half a million data points, each with 36 features

5.6.8 Image Manipulation

We use a blurring kernel (GF), which computes the 3 by 3 Gaussian blur for each pixel of the input image. Each warp is assigned an image tile consisting of 32 by 32 pixels, with threads being assigned a single row in the tile. The input is a randomly generated black and white image with 2048 by 2048 pixels resolution.

5.7 Evaluation

The baseline for all comparisons unless otherwise stated is that each core has a single warp.

5.7.1 Application Behavior

We first explore the performance characteristics and scaling behavior of the selected kernels on the baseline chip as outlined in Section 5.6.2. Table 5.3 shows some of the most important performance aspects of each application.

Each of these kernels access their main data structure in their critical loop, causing frequent cache misses due to low temporal or spatial locality. The number of instructions per memory operation is a good indicator of how well a given application will tolerate frequent cache misses.

K-means is the only kernel which can exploit the full performance of the base chip with only a single warp per core, achieving 2 teraops/sec. The other kernels are all limited by memory stalls to much lower performance, with sequence alignment achieving only 2.9% of the maximum possible performance.

Kernel Name	**instructions per memory op**	**off-chip bandwidth (GB/sec)**	**inst per sec. (MInst/sec)**
Neighbor List Generation (NL)	19	8.15	156
Lennard-Jones Force Calculation (LJ)	25	26.34	204
DNA Seq. Align. (SA)	7	62.77	57
Ray Tracing (RT)	15	30.60	124
K-Means (KM)	5	0.52	41
Gaussian Filter (GF)	8	77.91	65

Table 5.3: Number of instructions per memory operation, bandwidth usage and instructions per second for each kernel

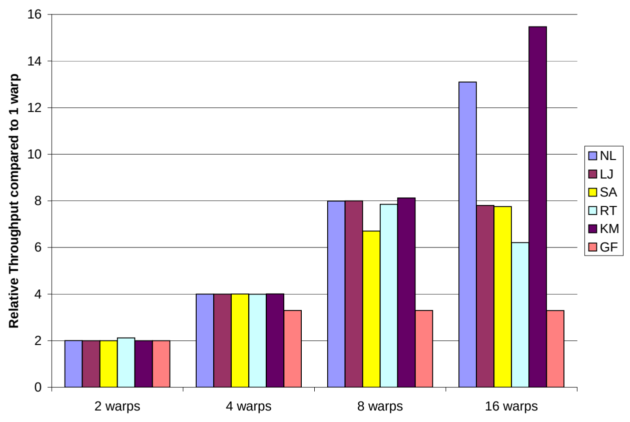

Figure 5.3: Increase in performance of 2 to 16 warps per core relative to a single warp per core.

Figure 5.3 shows the increase in throughput when the number of warps per core is increased from 1 to 16, and Figure 5.4 shows the bandwidth used for the same configurations. The neighbor list generation kernel shows the best increase, being limited by arithmetic throughput with 16 warps per core. On the other hand, the sequence alignment and ray tracing kernels become bandwidth bound at 4 and 8 warps respectively.

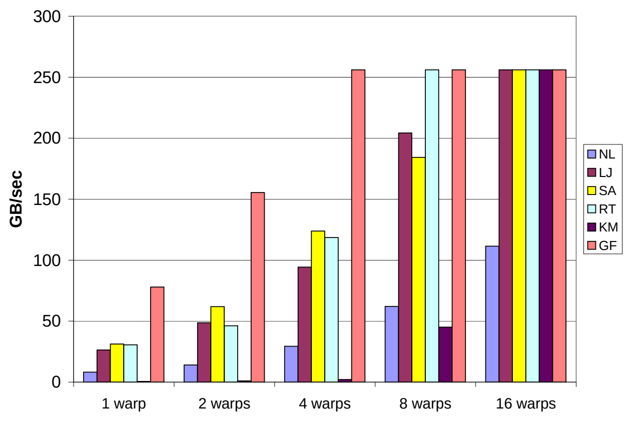

Figure 5.4: Bandwidth usage of all kernels with 1 to 16 warps per core. The total available bandwidth is 256 GB/sec.

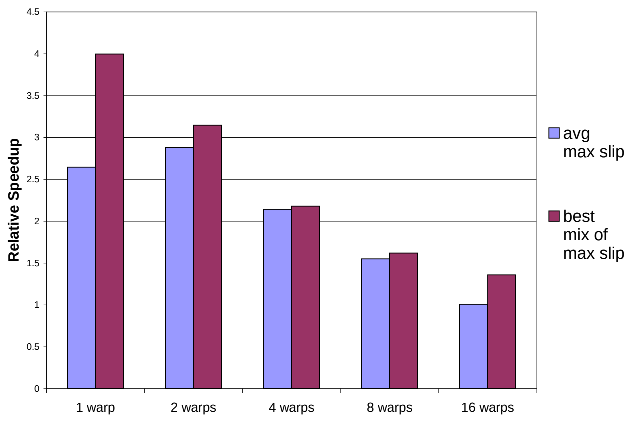

Figure 5.5: Relative speedup with 1 to 16 warps per core with a diverge on miss and a fixed maximum slip across all kernels for one particular config and combining the best fixed slip for each kernel.

5.7.2 Fixed Slip Performance

Figure 5.5 shows the relative speedup for 1 to 16 warps per core with a fixed maximum slip value compared to normal, blocking SIMD execution. We show both the speedup with the best average maximum slip across all kernels, as well as the speedup possible when combining the results with the best per-kernel fixed maximum slips. The difference at 1 and 2 warps is very significant, with relative speedups of of 2.65 vs. 4 at 1 warp per core and 2.88 vs. 3.15 at 2 warps per core. Moreover, the k-means kernel (which is ALU bound) exhibits a slowdown vs. blocking warps.

These results show clearly that the maximum slip value cannot be set statically across all applications, but has to adapt to the workload.

5.7.3 Adaptive Slip Performance

Figure 5.6 shows the speedup with the adaptive slip controller versus blocking warps for 1 to 16 warps per core.

The biggest gains can be seen for 1 to 4 warps per core, where the geometric mean speedup is 3.14 to 1.75 for 1 to 4 warps. At 8 and 16 warps per core many kernels become purely bandwidth bound, which means that improving latency per warp does not give any benefit. The 2D Gaussian filter has the highest speedup of all tested kernels, increasing throughput to 8.3 times the blocking warp implementation with 1 warp per core. As the number of warps increase the relative speedup compared to 1 warp per core decreases, as the kernel becomes bandwidth bound relatively quickly. The neighbor list generation and Lennard-Jones force calculation kernels also show high speedups at 2.5 and 5.6 respectively with 1 warp per core. As the number of warps per core is increased, the two kernels start to be limited by ALU throughput and bandwidth respectively, showing almost no speedup at 16 warps. The DNA sequence alignment kernel has its best speedup at 1 warp per core with 4.23. The alignment is also bandwidth bound as the number of warps increases.

K-means is a counterpoint to the other kernels, showing no appreciable speedup. This is because each thread reuses the data for the its point 32 times (once for each of the 32 cluster centers), leading to a small number of initial cache misses followed by the great majority of memory accesses hitting in the data cache. This behavior only changes at 8 and 16 warps, as the number of threads per core overwhelms the data cache and capacity misses result in a slowdown. With 8 warps, diverge on miss can provide a small speedup, as threads can reuse data in the cache in some cases where blocking warps would mean that the accesses would be too far apart in time. This is a good example how diverge on miss can help workloads which require a large number of warps for part of their execution, but are also limited by cache thrashing in other parts.

If I compare the performance of cores with diverge on miss to a core with a single warp and normal SIMD execution, diverge on miss increases performance by a factor of 3.14, 4.67, 5.38, 4.94 and 4.30. The peaking out at 4 warps is primarily due to cache thrashing kicking in on high warp counts. Compared to the base scaling shown in Figure 5.3, it can be seen that a core with 2 warps and diverge on miss can provide equivalent performance to a core with 16 warps and normal SIMD execution. From the area estimates in Section 5.5.1 we can see that such a core is approximately 35% smaller than a core with 16 warps.

Figure 5.7 shows the speedup across number of warp for both diverge on miss and normal execution. Diverge on miss with adaptive slip control provides a higher peak performance (5.38 times the baseline) than normal execution (4.14 times the baseline), but only requires 4 warps per core versus 16 warps per core. Because diverge on miss can tolerate more latency with a given number of warps, it is more limited by bandwidth limitations. As such, area saved by smaller cores could be used for more I/O, bigger caches or other structures which reduce off-chip bandwidth.

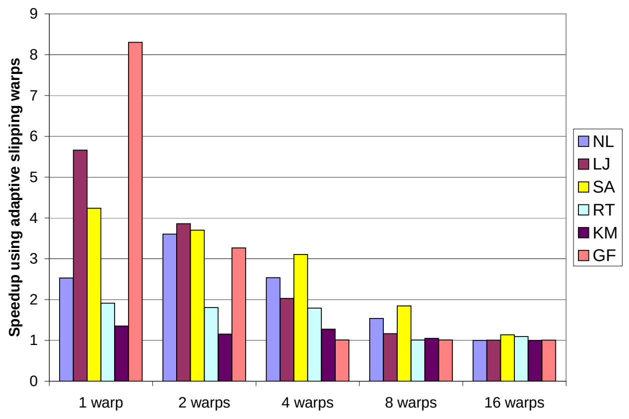

Figure 5.6: Speedup for 1 to 16 warps per core of adaptive slipping warps versus default blocking warps at the same number of warps per core.

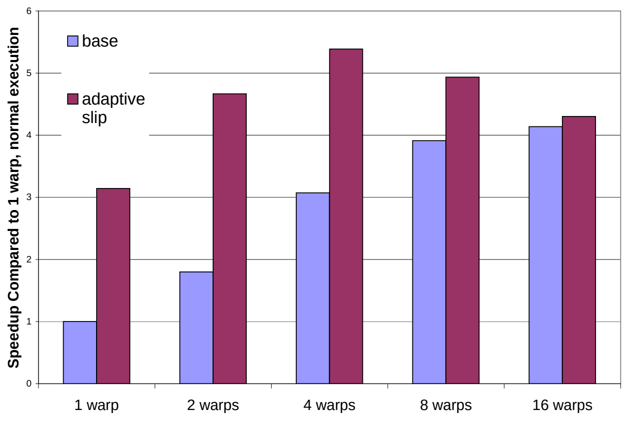

Figure 5.7: Comparing the speedup of both normal execution and slipping warps from 1 to 16 warps. The baseline is 1 warp per core with normal execution. Adaptive slip can provide a higher peak performance of 5.38 times the base performance versus 4.14 for normal execution, which needs 4 times more warps.

Since both Larrabee and Niagara provide unified second level cache on-chip, I also explore whether adding diverge on miss to a design where the SIMD cores are coupled to L2 caches is worthwhile. We simulate a design with the same number and type of cores as in previous experiments, but where each core has a private 256KB L2 cache. Each L2 cache is has 32B cache lines and is 16-way set associative. Off-chip bandwidth and access latency constant are kept constant from the previous experiments.

The mean performance of such a chip using normal SIMD execution is 8.1%,7.2%,7.4%,52.4% and 115.9% better than the chip without L2 caches for 1 to 16 warps per core and scaling from 1 to 16 warps per core improves from a factor of 4.14 to 6.35. The higher speedup at 8 and 16 warps per core is primarily due to several kernels making good use of the larger caches and being less bandwidth bound due to less cache thrashing. The relatively small gain for 1 to 4 warps is due to the fact that the NL, LJ and SA kernels suffer primarily from compulsory misses and no kernel exhibits cache thrashing with a small number of warps per core.

The relative speedup of using diverge on miss execution when each core has a 256KB L2 cache is shown in Figure 5.8. The performance increase is even larger than in Figure 5.6 primarily because kernels are less bandwidth bound and have fewer L2 caches misses, so that the misses which can be hidden with diverge on miss cover a larger fraction of all misses and provide a bigger relative improvement in performance.

5.8 Conclusion

To maximize performance within power and area constraints, designers have turned to architectures with many small, multithreaded SIMD cores for throughput oriented workloads. Such architectures work well for applications with regular data access patterns, but can easily become latency bound for workloads with more complicated scatter/gather

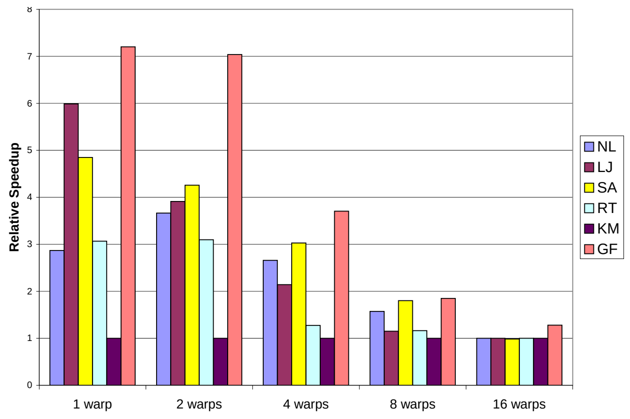

Figure 5.8: Speedup with adaptive slipping warps versus default blocking warps with each core having a private 256KB L2 cache.

access patterns.

We introduce the concept of diverge on miss, which allows SIMD warps to continue execution even when a subset of their threads are waiting on memory. This provides benefits when runahead threads prefetch cache lines for lagging threads. It also increases throughput when divergent threads experience relatively random misses and runahead and lagging threads continually leapfrog each other, rather than continually being held back by the slowest thread. Diverge on miss improves over my prior miss-divergence handling by requiring no additional warp scheduler entries and providing more robust speedups for workloads with complex memory access patterns. The key insight is that SIMD cores' support for branch divergence can be elegantly extended to support memory divergence, without having to re-group warps into finer grained scheduling units.

We show that on a set of data-parallel kernels, diverge on miss can provide speedups as high as 3.14 over normal SIMD execution or can reduce the core area by 35% at constant performance. It can also provide 30% higher absolute peak performance than normal execution with fewer warps per core.

Chapter 6

Sharing Tracker

6.1 Introduction

Graphics processing units (GPUs) were once fixed-function hardware for 3D rendering. Demand for increasing programmability for such applications have gradually driven GPU architectures to become general-purpose manycore architectures (embedded within a system-on-chip including various 3D-specific accelerators). The introduction of hardware and software support for general-purpose programming languages on the GPU [19, 77, 80] has allowed GPUs to become a viable platform for general-purpose computing.

Although the GPU instruction-set architecture is general-purpose, the memory hierarchy and performance model are different than traditional CPU architectures. GPU "cores" are deeply multi-threaded and wide array-style SIMD organizations. On-chip memory capacity is small. Together, these choices sacrifice single-thread performance in order to boost the number of cores and available memory bandwidth, optimizing for throughput instead.

GPU cores share global memory. Every core also possesses a "per-block shared memory" (PBSM) that is actually a software-controlled scratchpad. GPU cores also possess two small L1 data caches that were originally designed for specialized 3D-rendering access patterns (shared constants and texturing) that turn out to be useful for general-purpose workloads as well [16, 28, 110]. These caches are private and are not kept coherent. Values in these caches and in the PBSM must be kept coherent by software. This can be achieved by the programmer or the compiler (by flushing when necessary) and techniques for compiler-controlled software coherence have been studied for over 20 years (e.g., [29, 108]).

The private nature of these cores prevents re-using values shared among cores. Reuse reduces off-chip bandwidth requirements. Hardware coherence does capture reuse, at the expense of considerable complexity in order to support the correct semantics. Support for scalable hardware coherence has been studied for decades (Stenstrom [108] provides a good overview) and has recently been revisited for on-chip sharing [27, 47] in a multicore context. Since graphics workloads typically do not benefit from coherence, it is unlikely that GPUs will add the required hardware in the near future. The Cell BE [58] is another major general-purpose architecture that foregoes hardware coherence. Various multicore organizations for embedded systems also forego hardware coherence.

Capturing reuse with software-managed coherence requires some alternative means by which a core finds a cache line on a miss in its private L1 cache. One option is to have a last level cache (LLC) shared among all the cores. The drawback to such a design is that a LLC of sufficient size to support the request streams from a large number of wide SIMD cores will significantly reduce the chip area available for the high throughput cores.

Instead, I propose the *sharing tracker*, which simplifies the directory from cache coherence approaches for use with non-coherent cache hierarchies. The key insight is that when software is responsible for coherence, the directory becomes a predictor and a mere performance hint. Erroneous predictions may reduce performance but do not violate memory semantics. In contrast to full coherence directories, the sharing tracker is a low-cost structure that can be sized independently of the overall cache capacity it covers, and does not have the complexities associated with cache coherence protocols. A simplified directory-like sharing tracker is able to effectively capture reuse and fill misses from other private, on-chip caches. This greatly reduces off-chip accesses. With memory bandwidth increasingly becoming the limiting factor in throughput, this can have dramatic performance benefits.

On a set of memory intensive kernels the sharing tracker can increase performance by 5 to 12% for a manycore CMP where each core has a 32KB L1 and a 256KB L2 cache (8MB total L2 for a 32-core organization) and by 50 to 102% if those cores omit the L2 altogether and only have 32KB L1 caches. In fact, as long as the L1s have sufficient associativity, an L1-only organization with sharing tracking matches performance with the large L2. Eliminating the L2 can reduce cost or permit integration of additional cores. Adding the sharing tracker to a manycore CMP with only per-core caches can increase performance per mm^2 by 35%.

The effectiveness of the sharing tracker with only small per-core L1s is chiefly due to two factors. First, with many cores, the aggregate L1 capacity is still large (1 MB for 32 cores x 32 KB/core). Second, a latency-tolerant design converts the cache from a tool to reduce latency into a tool to conserve bandwidth. This means that cache misses have minimal cost as long as bandwidth is not a bottleneck. Of course, this requires sufficiently deep multi-threading to actually hide latency effectively. The sharing tracker's value is in capturing inter-core reuse that would otherwise have incurred off-chip accesses.

6.2 GPU Cache Architecture and Memory Model

I have given some background on GPU architecture in Section 1.5 and would like to talk here about the GPUs cache subsystems and the memory model.

GPU caches are specialized to deal with different address spaces and access patterns which are derived from the high level graphics APIs [13]. The question might be asked why GPUs have any caches for data at all, since they are optimized to tolerate latency.

The answer is that GPU caches are mostly meant as bandwidth savers and not as a way to decrease latency of memory accesses. To illustrate the caches in a GPU, I use the NVIDIA Tesla architecture [68] as an example, since it has the most publicly available information.

Each core has 3 main caches:

1. The instruction cache, which is the same as in a regular CPU. Code segments are effectively read-only, as there is no way for a GPU thread to change it at runtime.

2. The constant cache: This cache is meant for broadcasting values to all the SIMD threads. The data structures mapped to the address space of the constant cache are read-only and the cache doesn't support any form of writing. If different threads in a SIMD group request different values, the constant cache serializes the request. The latency of the constant cache is relatively low, and for good performance the workload has to exhibit temporal and spatial locality. As such, many of the design considerations for the constant cache are similar to the ones of a L1 data cache in a CPU.

3. The texture cache: This cache is meant for accessing textures, which in 3D graphics is the name given to images which are mapped onto triangles being rendered to the screen. Because of the nature of the graphics workload these caches act primarily to capture spatial locality in accesses from neighboring threads in a SIMD group. Hits in these caches have the same latency as misses, which limits their usefulness for many general-purpose workloads. Textures are also read-only.

It should be noted that the data structures cached in these caches can be modified, but this usually requires the intervention of the graphics card driver on the CPU and completely invalidating all the data in these caches.

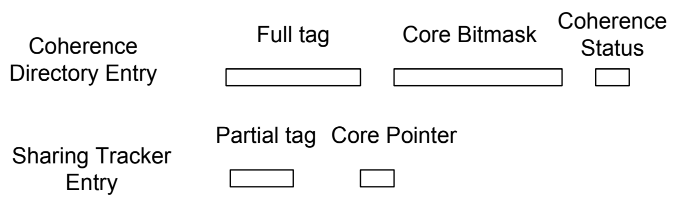

Figure 6.1: A cache coherence directory entry consists of a full tag, a bitmask indicating which cores have copies of a particular cache line and a small bitfield to track the current coherency state. In contrast, a sharing tracker entry consists of a smaller partial tag and a pointer to the cache that contains a particular cache line.

Since both constant and texture cache only support read-only data structures, many general-purpose workloads suffer from the fact that each access to a read/write data structure incurs the full latency of going to memory, which is several hundred cycles.

6.2.1 GPU Memory Model and its Implications

The GPU's memory model is that memory is non-coherent and there are no rules for ordering stores from a single core. Changes made by one core will only be guaranteed to be globally visible after a heavyweight global barrier, which basically involves flushing all the on-chip caches.

As I have mentioned in the prior section, all the current caches on a GPU only support read-only data structures. If caches which support for reads and writes are added, case of multiple cores writing to the same cache lines also has to be dealt with. I assume that such a chip will use a write-validate [57] policy to deal with this particular issue.

6.3 Adapting Coherency Hardware

Current GPUs have multiple SIMD cores, with small, per-core caches. To get better performance on general-purpose workloads I want to exploit sharing of cache lines between cores to reduce off-chip and latency of memory requests. One option would be to add a large, shared, inclusive LLC, which would naturally capture such re-use. But such a cache would occupy significant area, which might otherwise be devoted to more cores.

In traditional CMPs, cache coherency is used to figure out if there is a copy of a requested cache line in a cache on-chip and to request a copy. For manycore CMPs a snoopy coherency protocol would be problematic because of the rapid rise in communication volume as the number of cores increases. A directory protocol is the better choice for such an architecture. But of course, cache coherency does much more than that, ensuring that a core receives the most up to date version of a cache line and that if one core is writing to a cache line no other core has a valid copy.

This is too much functionality for my purposes, since I want to only save off-chip bandwidth and improve latency of memory requests. I want to decompose the functionality of directory-based cache coherency hardware and keep only the parts needed for my purposes.

- Tracking the status of cache lines (shared,exclusive,etc) is not necessary, since the current programming models of GPUs allow race and reads of stale data.
- Keeping track of all copies of a cache line is not necessary, since there is no constraint that a core must have the only copy of a cache line for a write.
- There can be cache lines which are on chip and not tracked at all. This is allowed since stale copies of cache lines are allowed. Any copying of cache lines between cores is simply to save off-chip bandwidth, not for correctness.

With these relaxations of the requirements versus full cache coherence I have derived a new structure from previous proposals for directory-based cache coherency hardware for CMPs [27, 47].

I call this new structure the *sharing tracker*.

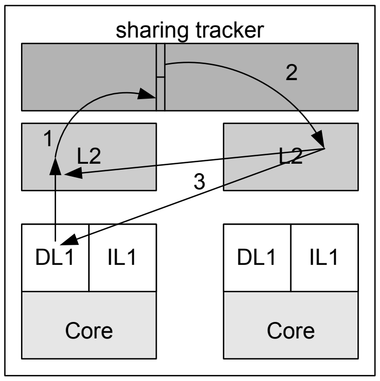

Figure 6.2: On an L2 miss, the request is sent to the sharing tracker (1). The sharing tracker is queried like a shared L3 cache. On a hit in the sharing tracker, it reads out the pointer in its entry and forwards the request to the appropriate L2 cache (2). If there is an L2 hit, a copy of the cache line is then forwarded to the original L2 and core (3).

6.3.1 Sharing Tracker Organization

Figure 6.1 shows the different units involved in a sharing tracker lookup. Note that in the following explanation I refer to all caches as being L2s, but of course the same mechanism applies if the GPU cores only have private L1 caches. The sharing tracker is organized like a shared cache, but each entry holds as data only a pointer to a private cache that contains the specific cache line. Unlike a full distributed coherence engine [47], the sharing tracker does not need to track all the cores which have a copy of a given cache line (which requires a bitmask which grows with the number of cores) or the current coherence state of a cache line.

When a L2 cache miss occurs, the sharing tracker is checked, similar to a shared L3 cache (see Figure 6.2). If there is a hit in the sharing tracker, a pointer to the cache holding that cache line (called the source cache) is read from the sharing tracker. A request is sent to the source cache. The source cache then does a normal cache lookup. Note that the lookup will not necessarily hit since the sharing tracker entry can be out of date or there was a false positive hit due to the use of a partial tag. If there is a hit in the source cache, that cache then forwards the cache line to the requesting cache. The sharing tracker's entry is updated to point to the requesting core. If a cache line is evicted from the private L2 cache of a core, the sharing tracker is checked for that entry. If the sharing tracker hits on that cache line AND the core id of the sharing tracker entry matches the L2 id from whose L2 the cache line is being evicted, the sharing tracker entry is invalidated. Note that it is possible that there are one or more copies of the evicted cache line in other private L2 caches on chip, which are lost for future sharing purposes if the corresponding entry in the sharing tracker is invalidated.

Unlike a distributed cache coherency directory [47], the sharing tracker does not have to return a correct prediction. Since each prediction is checked through an L2 lookup, false positives are caught automatically. If there is a miss in the source L2 cache, the request is sent to the memory subsystem and the corresponding sharing tracker entry is invalidated. If the sharing tracker lookup hits and returns the result that the source cache equals the requesting cache the checking logic knows immediately that a false positive has occurred, since the requesting cache has already done a lookup before sending the request to the sharing tracker.

Another advantage of not having to guarantee correct lookups is that by reducing the size of tags in the sharing tracker [36] (see Figure 6.1). As is shown in Section 6.5, it is possible to substantially reduce the tag size without unduly reducing the effectiveness of the sharing tracker. This is especially important for a cache-like structure such as the sharing tracker where the tag size can be larger than the data per entry.

6.4 Simulator

The general goal of my simulator is to let me explore new architectural ideas in the many-core space quickly. Since I observed previously that most programs on manycores are bound by the performance of the memory subsystem and because the manycore CMPs use very simple core architectures compared to traditional speculative, out-of-order cores, I have focused my efforts on the cache and memory subsystem while modeling instruction execution with the simplest model possible.

The custom simulator models a number of SIMD/vector cores, along with a cache hierarchy and a shared memory subsystem. The cores are modeled as having a constant CPI of one for all non-memory instructions, private L1 data caches, and the model assumes that the structures for holding outstanding memory requests are not a limiting factor. Each core can have one or multiple warps, and like current GPUs, can switch among warps on a cycle by cycle basis at no extra cost. The scheduling algorithm is round-robin, skipping warps which are waiting on memory requests. The memory reference traces are collected directly from the native applications, which are instrumented with calls to the simulator. Direct instrumentation of native applications was preferred over gathering large memory traces to avoid the I/O and decompression overheads of normal trace based simulators. To determine the number of instructions between memory references, each application is inspected manually and the number of arithmetic and control flow instructions between memory references are passed to the simulator.

The combination of a simple core model and direct instrumentation of native applications allows the simulator to be very fast (slowdowns of just 10-30x over pure native execution are the norm) and can consequently capture the performance on input sizes which would be prohibitively slow to simulate otherwise. This is especially important when dealing with a large number of cores and threads per core.

Number of cores	32
SIMD width	32
number of warps per core	1 - 16
Register File size per warp	4KB
Per core L1 instruction cache	32KB, 8-way, 64B lines
Per core L1 data cache	32KB, 8-way, 64B lines
Non-memory CPI	1
(Optional) per core L2 cache	256KB, 16-way, 64B lines
L2 hit latency	20 cycles
hit latency in remote L2 after lookup in sharing tracker	100 cycles
Off-chip bandwidth	256 GB/sec
memory latency	500 cycles
Clock speed	2 GHz

Table 6.1: Details of the simulated systems

6.4.1 Simulated System

The simulated system is described in Table 6.1. I assume a CMP consisting of 32 in-order cores each supporting 32-wide SIMD execution, all running at 2 GHz, for an overall maximum execution bandwidth of 2 Teraops. Each core has a 32KB private data cache, which has 64B cache lines and is 8-way set associative. I explore whether it makes sense to add a 256KB, 16-way set-associative L2 cache to each core (similar to the proposed Larrabee [97]) in terms of area efficiency or if having smaller cores with only L1 is enough. For all caches the simulator models a standard LRU replacement policy. I experimented with a variety of other replacement policies, e.g. adaptations of Qureshi's work [87, 86], with no major benefit. I assume that it takes 100 cycles to access the sharing tracker, forward the request to the source cache and copy a cache line to the requesting core's cache. All cores share a 256 GB/sec memory interface, with a memory access latency of 500 cycles.

Size of the physical address space supported	40 bits
size of full tag and valid bit	$15+1$ bits
size of bitmask and coherence state	$32+2$ bits
number of entries needed to cover 8MB of L2	128K
Total size of coherence directory	800KB
size of partial tag and valid bit	$10+1$ bits
size of L2 pointer	5 bits
Total size of sharing tracker covering 8MB	256KB

Table 6.2: Comparison of a coherence directory to the sharing tracker

core type	core area with only L1s(mm^2)	core area including a 256KB L2(mm^2)
32-wide SIMD core with 1 warp	7	11.35
32-wide SIMD core with 2 warps	7.3	11.65
32-wide SIMD core with 4 warps	7.9	12.25
32-wide SIMD core with 8 warps	9.1	13.45
32-wide SIMD core with 16 warps	11.5	15.85

Table 6.3: Area estimates for a different variants of a 32-wide SIMD core.

6.4.2 Area Model

To evaluate the tradeoff between additional cores or adding an L2 cache to each core or adding structures such as a sharing tracker I need an estimate of the chip area the different types of structures occupy. I use the same methodology as I described in Section 3.5.1 and 5.5.1 to arrive at the core sizes below.

I use Cacti 5 [103] to estimate the area of the per-core 256KB, 16-way set associative L2 cache as well as the other caches and cache-like structures. The area estimates from these calculations are shown in Table 6.3.

structure description	area (mm^2)
8MB LLC cache	44.65
full distributed coherence directory covering 8MB	3.42
sharing tracker covering 8MB	1.01
Area for inter-core network, IO-pads, etc.	74

Table 6.4: Area estimates cache-like structures and un-core.

For the calculations of area efficiency in Section 6.5 I also need an estimate for all the structures on a chip apart from the cores themselves. I estimate that the SIMD cores will occupy 75% of the die area, with the other 25% used for the inter-core network, IO-pads, memory buffers, etc. . I used the smallest SIMD core for this calculation and assumed as elsewhere that the chip would have 32 cores. The area of the cache-like structures and the non-core part of the chip are shown in Table 6.4.

I use the same workload for my evaluation as described in Section 5.6.3.

6.5 Evaluation

My initial investigations showed that reducing the tags to 10 bits showed no noticeable performance drop compared to full tags. I use 10 bit tags in all the following experiments.

6.5.1 Performance Comparison

To evaluate the overall impact of adding the sharing to a manycore GPU, I first show the performance and bandwidth improvement possible by adding a sharing tracker to the base CMP with per-core L2 as described in Section 6.4.1. In this first experiment, the L2s are maintained. The moderately large per-core L2s already capture much more of each core's working set (despite some duplication of data among L2s) than the 32 KB L1, so I expect modest benefit from adding the sharing tracker. I compute the geometric mean performance and bandwidth across all kernels introduced in Section 5.6.3, where the performance of each kernel in each configuration has been normalized to the performance of that kernel without a sharing tracker.

As can be seen in Figures 6.3 and 6.4, the sharing tracker can increase performance between 3 and 12% while reducing the required off-chip bandwidth by 20 to 45%. The kernels which benefit the most from the sharing tracker are those that are bandwidth bound

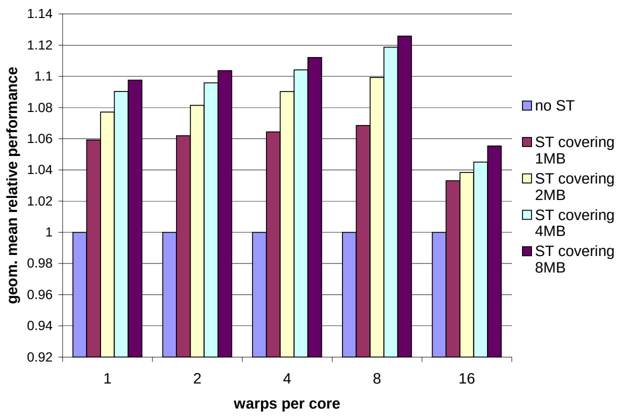

Figure 6.3: The geometric mean performance across all kernels using different sized sharing trackers(ST) (covering 0 to 8 MB of L2 cache entries) normalized to no sharing tracker. The number of warps per core is varied from 1 to 16.

and have significant sharing of data between threads on different cores. These are primarily the RT and SA kernels which both traverse very large data structures which are shared between all threads and have sharing complex patterns where widely spaced threads can access the same data. The KM, GF and NL generation kernels show no performance improvement. This is expected, as the GF and NL kernels have only local sharing of data which can be satisfied by each core's L2 caches. The KM kernel only shares a very small array between all threads and each thread touches only its private data apart from the very small global array, meaning it has no re-use which cannot be captures by the L1 caches.

I now evaluate the performance and bandwidth savings if each core only has L1 caches. Figures 6.5 and 6.6 show the performance and bandwidth improvements possible by adding a sharing tracker covering part or all off the L1 data caches. As the Figure shows, both the performance and bandwidth improvements are greater than when each core has an L2 cache. Performance improves between 50% and 102% relative to the L1-only case without the sharing tracker. The difference to the prior case is due to most kernels becoming much more bandwidth and latency bound. The RT, SA and LJ kernels show bigger improvements, but the real difference is that the KM and GF kernels now also improve in

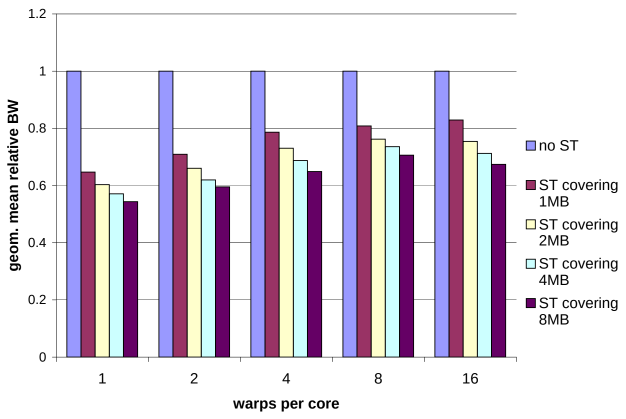

Figure 6.4: The geometric mean off-chip bandwidth across all kernels using different sized sharing trackers(ST) (covering 0 to 8 MB of L2 cache entries) normalized to no sharing tracker. The number of warps per core is varied from 1 to 16.

performance for some configurations. This is primarily the case because these kernels thrash their L1 caches at higher warp counts. Bandwidth savings are between 38 and 58% for similar reasons.

Clearly the individual L1s do not have sufficient capacity to capture each core's working set. Next I compare organizations with and without the L2. Figure 6.7 compares geometric mean performance across all kernels normalized to the performance of the configuration with the smallest chip area, which is one warp per core and no L2.

A first fact to note is that if the performance of cores with and without L2 cache and no sharing tracker is compared, the relative performance benefit of L2 grows as the number of warps per core is increased. This is due to the fact that more warps per core put more pressure on the caches and the L1 caches start to thrash for some kernels at 8 and 16 warps per core. It is very interesting to note that the small sharing tracker can lift the performance of the no-L2 configuration to the level of the configuration with L2. It is not clear how much this is due to limited long-range temporal locality in the suite of kernels, and how much due to latency tolerance with sufficient number of warps. At 16 warps, the L1-only organization with the best sharing tracker outperforms the conventional organization with

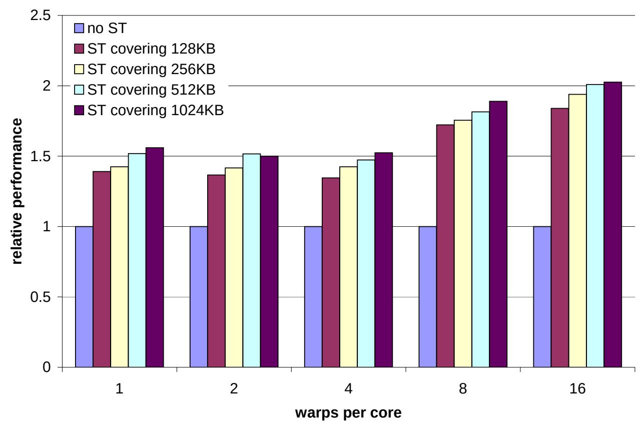

Figure 6.5: The geometric mean performance across all kernels using different sized sharing trackers(ST) (covering 0 to 1 MB of L1 cache entries) normalized to no sharing tracker. The number of warps per core is varied from 1 to 16.

256 KB L2 per core and no sharing tracker, and is within 2.4% of the configuration with L2 and the largest sharing tracker.

6.5.2 Performance/Area Comparison

Raw performance is not the only metric architects care about. Figure 6.8 shows the performance per mm^2 of each configuration. For this calculation I use the area of each core configuration from Table 6.3 and add the fixed overhead of the non-core part of the chip as shown in Table 6.4. Here it can be seen that in the base case (no sharing tracker) adding L2 caches to each core makes little sense even without the sharing tracker below 8 warps per core, as performance per mm^2 of the configurations with and without L2 caches are within 0 to 7% for 1 to 4 warps per core. At 8 warps per core that difference grows to 28% and to 50% at 16 warps.

With the addition of the sharing tracker, the performance per mm^2 of the configurations without L2 cache rises much more than of those with, making that configuration the top choice. The advantage is 40% with 1 warp, 26% with 2, 39% with 4, 31% with 8 and 28%

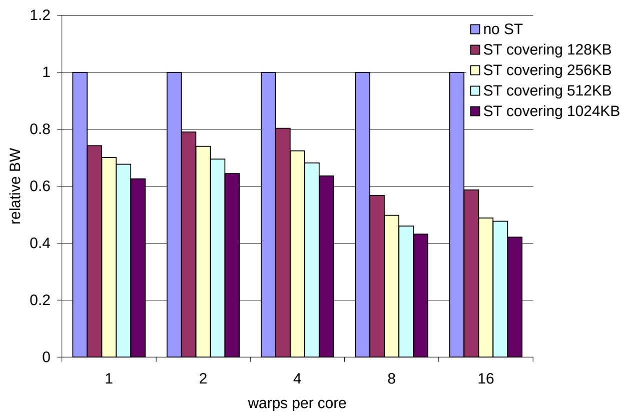

Figure 6.6: The geometric mean off-chip bandwidth across all kernels using different sized sharing trackers(ST) (covering 0 to 1 MB of L1 cache entries) normalized to no sharing tracker. The number of warps per core is varied from 1 to 16.

with 16 warps. Comparing the configuration with the highest performance per mm^2 (16 warps per core and per core L2 caches) without the sharing tracker to the one with sharing tracker (16 warps per core, no L2 caches, sharing tracker covering all of the on-chip L1 capacity), there is a 35% improvement.

To more clearly illustrate the benefit of removing the L2's from each core I plot the ratio of performance/mm^2 for each configuration with and without L2 caches and with different sized sharing trackers in Figure 6.9.

6.6 Conclusions and Future Work

GPUs have recently emerged as a new platform for high-performance computing. Their current cache organization is optimized for streaming data with little temporal locality and no sharing between cores, and requires software to manage any coherence requirements. To efficiently support general-purpose workloads, better support for temporal reuse is needed. However, as long as GPUs' main sales volume remains biased toward 3D rendering applications, and these do not require cache coherence, I think it is unlikely that

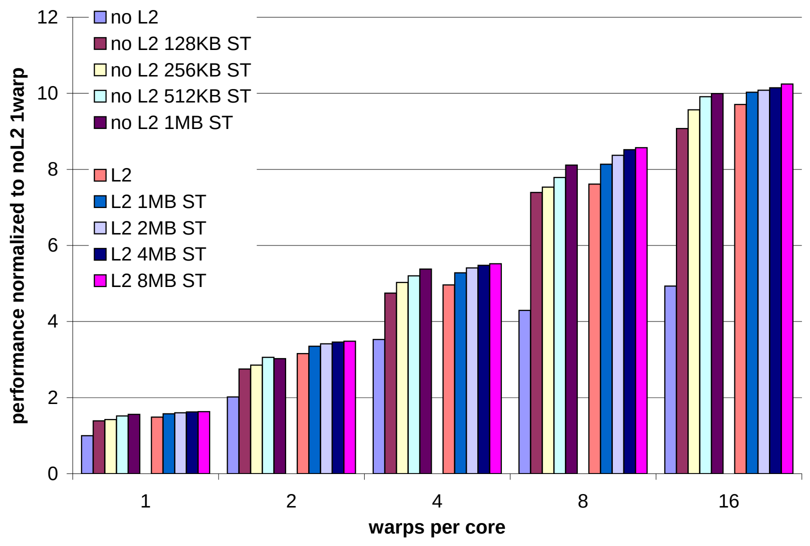

Figure 6.7: Performance comparison of cores with L2s and without. Performance is the geometry mean across all kernels normalized to the performance of 1 warp per core and no L2 cache or sharing tracker. The former 5 bars per configuration are without L2 and the latter 5 bars are with. The first bar of each group is with sharing tracker turned off and the rest show performance with sharing trackers of various coverage.

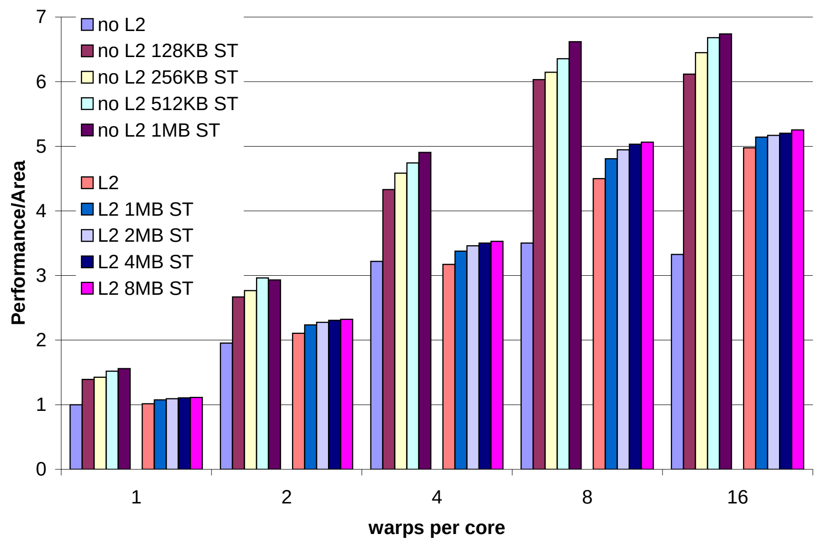

Figure 6.8: Performance per mm^2 for cores with and without L2 caches as the number of warps per core and the sharing tracker are scaled up.

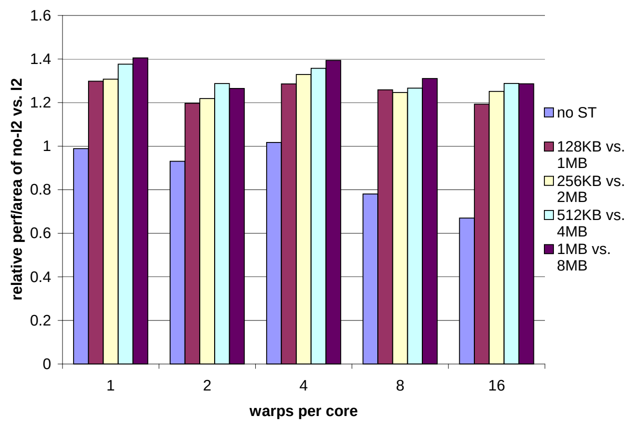

Figure 6.9: The ratio of the performance per unit area of configurations with and without per core L2 cache.

GPU vendors will add full cache coherence in the near future. Since cache coherence is challenging to implement at large scales in any case, software coherence is an appealing option for any large-scale multiprocessor, and some other organizations, notably Cell and RAW, have followed this approach.

This chapter shows that in a throughput-oriented processor with effective latency-tolerance mechanisms, a lightweight alternative (called the sharing tracker) to a full on-chip cache coherency directory provides all of the benefits of cache coherence for sharing cache lines among multiple caches on a chip, with 4 to 20 times less area than a coherent organization. The sharing tracker allows the L2 to be eliminated entirely and still boosts performance by 3%. Even in the case where the L2 organization includes a sharing tracker, the L1-only organization with sharing tracker only sacrifices 2.4%. The sharing tracker also reduces off-chip bandwidth by 38–58% compared to the over a conventional L2 organization, and increases performance per mm^2 by 35% compared to a design with only per-core caches. Although my results are obtained with a GPU organization, the success of the sharing tracker in that context suggests that other throughput oriented architectures should evaluate a similar approach. Generalizing this approach poses an interesting direction for future work.

As manycore CMPs increase the number of cores per chip, the latency of accessing any global structure will worsen. One way to deal with this problem is by replicating resources, but this is expensive for large structures. I want to investigate whether I can use the fact that the sharing tracker is small and does not require precise or up-to-date data, for a design which distributes multiple copies of the global sharing tracker across a CMP. This can reduce global on-chip network bandwidth and latency, potentially increasing performance. Another option I want to investigate is whether a hierarchical sharing tracker, where smaller sharing trackers cover a subset of cores and can resolve misses before they have to go to a global sharing tracker, can achieve the same bandwidth and latency advantages as the replicated sharing tracker.

Chapter 7

Conclusions

The design of microprocessors has traditionally focused on improving single-thread performance. This was accomplished by taking advantage of the increasing number of transistors per die due to Moore's Law and the increasing switching frequency of those transistors with each process generation. The larger number of transistors were used to design ever larger, more complex cores, which could execute more instructions per cycle with each generation. The size of on-chip caches was also rapidly increased, reducing the fraction of memory accesses which accessed off-chip memory and stalled the processor. The higher switching frequency of the transistors allowed new cores to run at a higher clock rate. Clock rates were further increased by increasing the pipeline depth of processors with each new design, which meant that each each pipeline stage did less work and had less delay, allowing new designs to be clocked higher, independent of the process generation that they were built in. The combination of more capable cores, deeper pipelining, and increasing switching speed of the underlying transistors led to a steady growth in single-thread performance for two decades [9].

Multiple factors led to the demise of this fortuitous combination. Computer architects had used several techniques to increase performance per clock cycle over time. By the

1990s, they had turned to out-of-order execution, which is a limited form of data-flow execution combined with bookkeeping structures to make the result appear as if the processor had executed the program in-order, so as not to complicate the abstract processor model that programmers had to deal with. While out-of-order cores initially provided good performance increases, their efficiency decreased steadily as they were pushed to higher and higher performance levels. Finally, further performance improvements were only possible at unacceptable cost. Computer architects therefore relied much more heavily on increasing the pipeline depth of processors, hoping that a faster increase in the achievable clock rate could offset the constant or decreasing performance per cycle. Decreasing performance is possible because aggressively increasing the pipeline depth of a core leads to lower performance per clock. The Intel Pentium 4 [14] is an example of a design which traded slightly lower per-clock performance for much higher clock speed. A consequence of increasing the clockrate faster than simple process scaling allowed was that the power consumption of microprocessors increased significantly. Rising power consumption due to aggressively increased clock speeds was one of the reasons that the performance growth of single-core microprocessors slowed dramatically, but not the only one.

The model for ideal scaling of CMOS process technology was introduced by Dennard [33]. One important feature of Dennard scaling is that it keeps the power consumption of a fixed size chip constant across process generations. At its core was the assumption that the supply voltage of a chip and the threshold voltage of the transistors could be scaled down each process generation, thereby compensating for higher frequency. The scaling of threshold voltage has dramatically slowed down or even stopped in recent years, due to the limits subthreshold leakage sets on threshold voltage. This in turn limits the further lowering of the supply voltage and leads to increasing, not constant, power consumption as a chip runs at a higher frequency at a newer process node. The higher power consumption from both aggressive pipelining and the slowing of supply voltage scaling severly limited the increase in performance that newer designs could provide.

Once the performance growth of single-core microprocessors slowed, the microprocessor industry turned to integrating multiple processor cores in a single processor to further increase performance. Having multiple cores per die meant that higher clock rates were not required for higher throughput. In fact, two or more cores could be run at a lower frequency and voltage, and potentially lower power, and still provide higher throughput than a single core. Note that, unlike previous performance increases of microprocessors, scaling the number of cores per chip requires programmers to actively change their programs to exploit more and more parallelism to obtain the full benefit of the increased throughput of such chip multiprocessors.

The design of CMPs has been an active research topic. Prior work [49, 67] has shown that asymmetric CMPs, which have a small number of large, high-performance cores and a large number of small, throughput-oriented cores, can outperform homogeneous CMPs, where all cores are of the same type. The large, high-performance cores use the same type of design as the last generation of single-core processors, which are both power- and area-inefficient. The throughput-oriented cores are much smaller but individually provide significantly lower performance, leaving a significant gap and leading to sub-optimal performance for many workloads.

To narrow the gap in performance requires building cores which are higher-performance than throughput cores, but have higher area-efficiency. In Chapter 3, I showed that an out-of-order core can be built with much smaller, more power efficient structures than previously thought at only a slight loss of performance and a gain in both energy- and area-efficiency [113, 114, 115].

The design of the new, lightweight structures, the consumer-based issue queue and the memory alias table, take advantage of several properties of typical programs when execute on a speculative out-of-order core.

1. The average number of consumers of a value produced by an instruction is close to one. Even for instructions which have a large number of consumers, the probability that a large number of them will be in the instruction window at the same time is limited. This property of the data-flow graphs of most programs means that traditional issue queues, which allow a single instruction to have $N-1$ consumers in the issue queue (where N is the size of the issue queue), are largely overdesigned, even for aggressive cores.

2. The probability that loads read values produced by stores that are still in the pipeline of a processor is small, and performance of small and medium sized cores does not suffer significantly if the mechanism to deal with such occurrences is slow. It is enough to have a very small hardware structure which can catch all such occurrences, even if there is a small probability of false positives, as long as there are no false negatives.

This new, lightweight out-of-order core has performance only 6.5% lower than a traditional 2-way out-of-order core, while using 22% less area.

Because of differences between and within parallel programs, the optimal number and size of throughput cores per chip will vary. An ideal future CMP could dynamically combine processing elements into different sizes cores depending on the need of the running program [49].

While the process of combining an arbitrary number of cores in a single, larger core seems infeasible, I showed in Chapter 4 how to combine two, throughput-oriented, multithreaded, scalar cores into a single larger, higher-performance out-of-order core, using the lightweight structures previously introduced [113, 114, 115]. This technique, called Federation, uses the fact that cores on a chip multiprocessors are very close and can have very low latency communication, as well as the fact that the large register files of multithreaded throughput cores can be re-used for the largest buffer structure of an out-of-order core, the active list. Federation can provide single-thread performance 92.4% higher than any single core it is built from, provide the full throughput of its constituent cores if needed, yet only adds a 3.7% area overhead.

Once programs are written to express their computation in a parallel manner to take advantage of CMPs, this opens up new options for architects. One example is that parallel programs can run not just on multiple scalar cores, but on cores with SIMD instruction sets.

SIMD cores have the advantage over scalar cores that they amortize the area and power of a single core frontend, usually understood to be the instruction cache, fetch, decode and control logic units, over multiple backends, which consist of the register file, execution units, and data caches. Compared to a CMP consisting of scalar cores, a CMP using SIMD cores can be either smaller and lower power, and have the same peak throughput, or the same size and power, and have higher peak throughput.

SIMD cores also have several limitations, which can prevent them from reaching their full potential. For example, programs with irregular memory access patterns often have low performance on SIMD cores, as the lockstep nature of SIMD execution forces many threads to wait on the minority of threads which incurred data cache misses.

In Chapter 5 I proposed a mechanism, called Diverge on Miss, that allows SIMD cores to continue to execute even if a subset of their threads are waiting on memory. The key insight that made Diverge on Miss feasible is that it can re-use the control logic which is already in place in SIMD cores to deal with branch divergence, which occurs when threads on a SIMD core do not all follow the same control-flow path, and only needs to add one small new structure and make incremental changes to another structure. Diverge on Miss can increase performance by 30% compared to a design with normal SIMD cores.

Another limiting factor for SIMD architectures is the performance of the memory subsystem. For CMPs using SIMD cores without cache coherency, such as modern graphics processing units, there is a need for a way cores can find and re-use cache lines in other cores caches to reduce off-chip bandwidth.

In Chapter 6, I showed how to build an imprecise directory, called the sharing tracker, at low cost. The sharing tracker takes advantage of the fact that in a non cache-coherent system a directory lookup can produce wrong or out-of-date information, since it is simply a performance hint and not required for correctness. The sharing tracker can improve performance per unit area by 35%, primarily by allowing smaller caches per core while delivering performance better than a configuration with larger caches.

Overall, this dissertation has focused on architectural techniques to improve the performance and efficiency of small cores of future asymmetric chip multiprocessors. This work will help future multicore microprocessors achieve higher and more robust performance for a wide variety of workloads, with lower power and smaller area.

7.1 Challenges for Manycore Architecture Research

The last five years have been truly momentous in computer architecture research. The dominance of the sophisticated, speculative out-of-order architectures has given way to the rise of multicore and then manycore architectures, many of which use cores which consciously eschew all the new structures and techniques invented for out-of-order cores over the last twenty years. Whole fields of inquiry which were more or less abandoned by the mainstream of academic research in the early 1990s are once again relevant.

Overshadowing all of this excitement is the uncertainty that nobody knows what software future multicore and manycore processors will run. In fact, nobody even knows what the dominant programming model will be. Many of the applications which are widely used on today's desktops do not seem to benefit from much more computing power for their current functionality. Most of today's benchmarks are single-threaded and cannot be used for judging the qualities of any proposed multicore architecture.

To find applications which architecture researchers can use today, they must look for application domains which already have massive parallelism expressed in their programs. These include many server workloads, which serve many users at the same time and can spawn a thread for each user. Traditional high-performance computing programs, which have been already extensively parallelized to run on supercomputers, are also capable of using a large number of cores and threads. Computing domains which have traditionally used specialized processors, such as graphics, video and many signal processing workloads, can also use many cores efficiently. In fact, it is possible that the mainstream use of manycore processors with small, efficient cores will lead to these niches to re-adopt mainstream processors to some degree.

All of these domains have many interesting and hard problems, but which (if any) will be a major part of future workloads? Without a clear idea of what the workload mix will look like, researchers are faced with a chicken and egg problem. Without a set of benchmark programs, it is impossible to meaningfully evaluate any proposed multicore design. I was faced with this problem when doing my work on SIMD manycore processors. I had to search quite widely to find the few programs that I did. It took many years for industry and academia to come to a consensus, embodied in the SPEC CPU benchmark suite, on what the right set of benchmarks were to measure the performance of single-threaded processors. It is possible that it will take equally long for a consensus to emerge for multicore processors. Until such a time, researchers in the field of computer architecture will have to do their own gathering of programs and exploring their characteristics.

Bibliography

[1] Intel Advanced Vector Extensions Programming Reference.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259, 2000.

[3] K. Aingaran, P. Kongetira, and K. Olukotun. Niagara: a 32-way Multithreaded Sparc Processor. *Micro, IEEE*, 25, 2005.

[4] Yoav Almog, Roni Rosner, Naftali Schwartz, and Ari Schmorak. Specialized Dynamic Optimizations for High-Performance Energy-Efficient Microarchitecture. In *2nd International Symposium on Code Generation and Optimization*, page 137, 2004.

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *ICS '90: Proceedings of the 4th international conference on Supercomputing*, pages 1–6, 1990.

[6] AMD. ATI CTM Guide: Technical reference manual. Technical report, AMD, 2006. Version 1.01.

[7] AMD. ATI Radeon HD 2900 Technology GPU Specifications, 2007.

[8] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations fully implemented on Graphics Processing Units. *J. Comput. Phys.*, 227(10):5342–5359, 2008.

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[10] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. pages 163–174, 2009.

[11] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture based on Single-Chip Multiprocessing. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, 2000.

[12] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *21st International Symposium on Computer Architecture*, May 1994.

[13] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[14] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah T. Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8:1–17, 2003.

[15] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[16] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2009.

[17] E. Brekelbaum, II Rupley, J., C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *35th International Symposium on Microarchitecture*, pages 27–36, 2002.

[18] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *27th International Symposium on Computer Architecture*, 2000.

[19] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, , and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[20] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[21] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[22] J. Adam Butts and Gurindar S. Sohi. Characterizing and Predicting Value Degree of Use. In *35th International Symposium on Microarchitecture*, pages 15–26, 2002.

[23] Alper Buyuktosunoglu, Ali El-moursy, and David H. Albonesi. An oldest-first selection logic implementation for noncompacting issue queues. In *15th International ASIC/SOC Conference*, pages 31–35, 2002.

[24] Brad Calder and Dirk Grunwald. Next Cache Line and Set Prediction. In *22nd International Symposium on Computer Architecture*, 1995.

[25] Doug Carmean. Future CPU Architectures: The Shift from Traditional Models. Intel Higher Education Lecture Series, 2007.

[26] J. B. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *In Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[27] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 264–276, 2006.

[28] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A Performance Study of General-Purpose Applications on Graphics Processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[29] H. Cheong and A.V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *ISCA '88: Proceedings of the 15th annual international symposium on Computer architecture*, pages 299–307, May 1988.

[30] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. *SIGARCH Comput. Archit. News*, 32(2):76, 2004.

[31] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with Streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, 2003.

[32] John D. Davis, James Laudon, and Kunle Olukotun. Maximizing CMP Throughput with Mediocre Cores. In *15th Conference on Parallel Architectures and Compilation Techniques*, 2005.

[33] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974.

[34] Romain Dolbeau and Andr&#233; Seznec. CASH: Revisiting Hardware Sharing in Single-Chip Parallel Processors. *J. of Instruction-Level Parallelism*, 6, 2004.

[35] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An Integrated Quad-Core Opteron Processor. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 102–103, Feb. 2007.

[36] Barry Fagin. Partial Resolution in Branch Target Buffers. *IEEE Trans. Comput.*, 46(10):1142–1145, 1997.

[37] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *MICRO 30: Proceed-*

ings of the 30th annual ACM/IEEE international symposium on Microarchitecture, pages 149–159, 1997.

[38] Brian K. Flachs, Shigehiro Asano, Sang H. Dhong, H. Peter Hofstee, Gilles Gervais, Roy Kim, Tien Le, Peichun Liu, Jens Leenstra, John S. Liberty, Brad W. Michael, Hwa-Joon Oh, Silvia M. Müller, Osamu Takahashi, Koji Hirairi, Atsushi Kawasumi, Hiroaki Murakami, Hiromi Noro, Shoji Onishi, Juergen Pille, Joel Silberman, Suk-soon Yong, Akiyuki Hatakeyama, Yukio Watanabe, Naoka Yano, Daniel A. Brokenshire, Mohammad Peyravian, VanDung To, and Eiji Iwata. Microarchitecture and Implementation of the Synergistic Processor in 65-nm and 90-nm SOI. *IBM Journal of Research and Development*, 51(5):529–544, 2007.

[39] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007.

[40] Amit Gandhi, Haitham Akkary, Ravi Rajwar, Srikanth T. Srinivasan, and Konrad Lai. Scalable load and store processing in latency-tolerant processors. *IEEE Micro*, 26(1):30–39, 2006.

[41] Ilya Ganusov and Martin Burtscher. Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 144–153, 2006.

[42] Alok Garg, Fernando Castro, Michael Huang, Daniel Chaver, Luis Pinuel, and Manuel Prieto. Substituting Associative Load Queue with Simple Hash Tables in

Out-of-Order Microprocessors. In *ISLPED '06: Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, pages 268–273, 2006.

[43] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE MICRO*, 28(4):13–27, 2008.

[44] Andy Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Ideas*, 1998.

[45] Masahiro Goshima, Kengo Nishino, Toshiaki Kitamura, Yasuhiko Nakashima, Shinji Tomita, and Shin ichiro Mori. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 225–236, Washington, DC, USA, 2001. IEEE Computer Society.

[46] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of Both Latency and Throughput. In *ICCD '04: Proceedings of the 22nd International Conference on Computer Design*, pages 236–243, 2004.

[47] Enric Herrero, José González, and Ramon Canal. Distributed Cooperative Caching. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 134–143, 2008.

[48] Phil Hester. 2006 AMD Analyst Day Presentation, 2006.

[49] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*. To appear.

[50] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 5:1–13, 2001.

[51] H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *11th International Conference on High Performance Computer Architecture*, pages 258–262, 2005.

[52] M. S. Hrishikesh, Doug Burger, Norman P. Jouppi, Stephen W. Keckler, Keith I. Farkas, and Premkishore Shivakumar. The optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, 2002.

[53] Michael Huang, Jose Renau, and Josep Torrellas. Energy-Efficient Hybrid Wakeup Logic. In *ISLPED '02: Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, 2002.

[54] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A nuca substrate for flexible cmp cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, 2005.

[55] Engin İpek, Meyrem Kırman, Nevin Kırman, and José Martínez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *34th International Symposium on Computer Architecture*, 2007.

[56] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit Power Efficient Sparc Soc. In *ISSCC'07*, pages 2–2, 2007.

[57] Norman P. Jouppi. Cache Write Policies and Performance. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 191–201, 1993.

[58] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell processor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[59] R. Kalla, Balaram Sinharoy, and J.M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *Micro, IEEE*, 24(2):40–47, Mar-Apr 2004.

[60] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable Stream Processors. *IEEE Computer*, pages 54–62, August 2003.

[61] R.E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 Microprocessor Architecture. In *ICCD '98: Proceedings of the 16th International Conference on Computer Design*, 1998.

[62] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, 2002.

[63] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable Lightweight Processors. In *40th International Symposium on Microarchitecture*, 2007.

[64] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The visual instruction set (vis) in ultrasparc. *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers.*, pages 462–469, Mar 1995.

[65] Gurhan Kucuk, Oguz Ergin, Dmitry Ponomarev, and Kanad Ghose. Distributed reorder buffer schemes for low power. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, 2003.

[66] R. Kumar, N.P. Jouppi, and D.M. Tullsen. Conjoined-Core Chip Multiprocessing. In *37th International Symposium on Microarchitecture*, pages 195–206, 2004.

[67] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *31st International Symposium on Computer Architecture*, page 64, 2004.

[68] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[69] Gabriel H. Loh. The Cost of Uncore in Throughput-Oriented Many-Core Processors. In *Workshop on Architectures and Languages for Throughput Applications*, 2008.

[70] Milo M. K. Martin, Pacia J. Harper, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 206–217, 2003.

[71] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *Micro, IEEE*, 25(2):10–20, March-April 2005.

[72] Jiayuan Meng, David Tarjan, and Kevin Skadron. Leveraging Memory Level Parallelism Using Dynamic Warp Subdivision. Technical Report CS-2009-02, University of Virginia, 2009.

[73] Francisco J. Mesa-Martinez, Joseph Nayfach-Battilan, and Jose Renau. Power Model Validation Through Thermal Measurements. In *34th International Symposium on Computer Architecture*, 2007.

[74] Michael Mantor. Radeon R600, a 2nd Generation Unified Shader Architectur, 2007.

[75] Pierre Michaud, André Seznec, and Stéphan Jourdan. An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors. *Int. J. Parallel Program.*, 29(1):35–58, 2001.

[76] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), 1965.

[77] A. Munshi. The OpenCL specification, version 1.0, document revision 29, Dec. 2008.

[78] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *9th International Conference on High Performance Computer Architecture*, page 129, 2003.

[79] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188, 2006.

[80] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[81] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Technical report, NVIDIA Corporation, Feb. 2007. Version 0.8.

[82] Stuart Oberman, Greg Favor, and Fred Weber. Amd 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[83] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *ASPLOS-VII: Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[84] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[85] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4):42–50, Aug 1996.

[86] M.K. Qureshi. Adaptive Spill-Receive for robust high-performance caching in CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 45–54, 2009.

[87] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 381–391, New York, NY, USA, 2007. ACM.

[88] Matthias Raab, Leonhard Grünschloss, Johannes Hanikaz, Manuel Finckh, and Alexander Keller. bwfirt.

[89] Amir Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *32nd International Symposium on Computer Architecture*, 2005.

[90] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[91] N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, and A. Kovacs. The Implementation of the 65nm Dual-Core 64b Merom Processor. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 106–590, Feb. 2007.

[92] T. Sakurai and A.R. Newton. A simple MOSFET model for circuit analysis. *Electron Devices, IEEE Transactions on*, 38(4):887–894, Apr 1991.

[93] Pierre Salverda and Craig Zilles. Fundamental Performance Challenges in Horizontal Fusion of In-Order Cores. In *14th International Conference on High Performance Computer Architecture*, page ??, 2008.

[94] Peter G. Sassone, Jeff Rupley II, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black. Matrix Scheduler Reloaded. In *34th International Symposium on Computer Architecture*, 2007.

[95] Toshinori Sato, Yusuke Nakamura, and Itsujiro Arita. Revisiting direct tag search algorithm on superscalar processors. In *in Proc. of Workshop on ComplexityEffective Design, held in conjunction with ISCA28*, 2001.

[96] Michael Schatz, Cole Trapnell, Arthur Delcher, and Amitabh Varshney. High-Throughput Sequence Alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[97] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[98] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *36th International Symposium on Microarchitecture*, page 399, 2003.

[99] Andr&#233; Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *29th International Symposium on Computer Architecture*, pages 295–306, 2002.

[100] Tingting Sha, Milo M. K. Martin, and Amir Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *38th International Symposium on Microarchitecture*, pages 159–170, 2005.

[101] Tingting Sha, Milo M. K. Martin, and Amir Roth. NoSQ: Store-Load Communication without a Store Queue. In *39th International Symposium on Microarchitecture*, pages 285–296, 2006.

[102] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[103] Shyamkumar Thoziyoor and Naveen Muralimanohar and Jung Ho Ahn and Norman P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[104] Dezsö Sima. The Design Space of Register Renaming Techniques. *IEEE Micro*, 20(5):70–83, 2000.

[105] Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn McKinley. Compiling for EDGE Architectures. In *4th International Symposium on Code Generation and Optimization*, 2006.

[106] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 521–532, New York, NY, USA, 1998. ACM.

[107] B. Stackhouse, S. Bhimji, C. Bostak, D. Bradley, B. Cherkauer, J. Desai, E. Francom, M. Gowan, P. Gronowski, D. Krueger, C. Morganti, and S. Troyer. A 65 nm 2-Billion Transistor Quad-Core Itanium Processor. *Solid-State Circuits, IEEE Journal of*, 44(1):18–31, Jan. 2009.

[108] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *Computer*, 23(6):12–24, 1990.

[109] B. Stolt, Y. Mittlefehldt, S. Dubey, G. Mittal, M. Lee, J. Friedrich, and E. Fluhr. Design and Implementation of the POWER6 Microprocessor. *Solid-State Circuits, IEEE Journal of*, 43(1):21–28, Jan. 2008.

[110] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 261–272, 2008.

[111] Samantika Subramaniam and Gabriel H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *39th International Symposium on Microarchitecture*, pages 273–284, 2006.

[112] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 253–264, 2009.

[113] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: Very low overhead Out-of-Order Execution. *ACM Transactions on Architecture and Code Optimization*, accepted pending major revisions.

[114] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: Out-of-Order Execution using Simple In-Order Cores. Technical Report CS-2007-11, Dept. of Comp. Sci., Univ. of Virginia, Aug. 2007.

[115] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue. In *Proceedings of the 45th annual Conference on Design Automation (DAC)*, pages 772–775, 2008.

[116] S. Thakkur and T. Huff. Internet streaming simd extensions. *Computer*, 32(12):26–34, Dec 1999.

[117] Marc Tremblay and J. Michael O'Connor. UltraSparc I: A Four-Issue Processor Supporting Multimedia. *IEEE Micro*, 16(2):42–50, 1996.

[118] Jessica H. Tseng and Krste Asanovic. RingScalar: A Complexity-Effective Out-of-Order Superscalar Microarchitecture. Technical Report MIT-CSAIL-TR-2006-066, MIT CSAIL, Sep. 2006.

[119] L.W. Tucker and G.G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–38, Aug 1988.

[120] 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. Baring It All to Software: Raw Machines. *Computer*, 30(9):86–93, 1997.

[121] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending Multicore Architectures to Exploit Hybrid parallelism in Single-thread Applications. In *13th International Conference on High Performance Computer Architecture*, 2007.