A

A Hierarchical Thread Scheduler and Register File for Energy-efficient Throughput Processors

MARK GEBHART, The University of Texas at Austin
DANIEL R. JOHNSON, University of Illinois at Urbana-Champaign
DAVID TARJAN, NVIDIA
STEPHEN W. KECKLER, NVIDIA and The University of Texas at Austin
WILLIAM J. DALLY, NVIDIA and Stanford University
ERIK LINDHOLM, NVIDIA
KEVIN SKADRON, University of Virginia

Modern graphics processing units (GPUs) employ a large number of hardware threads to hide both function unit and memory access latency. Extreme multithreading requires a complex thread scheduler as well as a large register file, which is expensive to access both in terms of energy and latency. We present two complementary techniques for reducing energy on massively-threaded processors such as GPUs. First, we investigate a two-level thread scheduler that maintains a small set of active threads to hide ALU and local memory access latency and a larger set of pending threads to hide main memory latency. Reducing the number of threads that the scheduler must consider each cycle improves the scheduler's energy efficiency. Second, we propose replacing the monolithic register file found on modern designs with a hierarchical register file. We explore various tradeoffs for the hierarchy including the number of levels in the hierarchy and the number of entries at each level. We consider both a hardware-managed caching scheme and a software-managed scheme, where the compiler is responsible for orchestrating all data movement within the register file hierarchy. Combined with a hierarchical register file, our two-level thread scheduler provides a further reduction in energy by only allocating entries in the upper levels of the register file hierarchy for active threads. Averaging across a variety of real world graphics and compute workloads, the active thread count can be reduced by a factor of 4 with minimal impact on performance and our most efficient three-level software-managed register file hierarchy reduces register file energy by 54%.

Categories and Subject Descriptors: C.1.4 [**Computer Systems Organization**]: Processor Architectures – *Parallel Architectures*

General Terms: Experimentation, Measurement

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
This article is based on the following two papers: "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors" presented at the International Symposium on Computer Architecture (ISCA) 2011 and "A Compile-Time Managed Multi-Level Register File Hierarchy" presented at the International Symposium on Microarchitecture (MICRO) 2011. We extend prior work by integrating the two approaches into a single evaluation showing the tradeoffs between hardware, software, and hybrid control approaches to managing a hierarchical register file.
Author's addresses: M. Gebhart, Computer Science Department, The University of Texas at Austin; D. Johnson, Department of Electrical and Computer Engineering, The University of Illinois at Urbana-Champaign; D. Tarjan and S. Keckler and W. Dally and E. Lindholm, NVIDIA Corporation, Santa Clara, CA; K. Skadron, Computer Science Department, University of Virginia; email: mgebhart@cs.utexas.edu.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Additional Key Words and Phrases: Energy efficiency, multi-threading, register file organization, throughput computing

ACM Reference Format:
Gebhart, M., Johnson, D., Tarjan, D., Keckler, S., Dally, W., Lindholm, E., Skadron, K. 2011. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. ACM Trans. Comput. Syst. V, N, Article A (January YYYY), 38 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Graphics processing units (GPUs) such as those produced by NVIDIA [NVIDIA 2009] and AMD [AMD 2011] are massively parallel programmable processors originally designed to exploit the concurrency inherent in graphics workloads. Modern GPUs contain thousands of arithmetic units, tens of thousands of hardware threads, and can achieve peak performance in excess of several TeraFLOPS. Many non-graphics workloads with high computational requirements also can be expressed as a large number of parallel threads. These workloads can take advantage of the parallel resources of modern GPUs to achieve high-throughput performance. At the same time, improvements in single-thread performance on CPUs have slowed and per-socket power limitations restrict the number of high-performance CPU cores that can be integrated on a single chip. Thus, GPUs have become attractive targets for achieving high performance across a wide class of applications.

Unlike CPUs that generally target single-thread performance, GPUs target high throughput by employing extreme multithreading [Fatahalian and Houston 2008]. For example, NVIDIA's Fermi design has a capacity of over 20,000 threads interleaved across 512 processing units [NVIDIA 2009]. Just holding the register context of these threads requires substantial on-chip storage – 2 MB in total for the maximally configured Fermi chip. Further, extreme multithreaded architectures require a thread scheduler that can select a thread to execute each cycle from a large hardware-resident pool. Accessing large register files and scheduling among a large number of threads consumes precious energy that could otherwise be spent performing useful computation. As existing and future integrated systems become power limited, energy efficiency (even at a cost to area) is critical to system performance.

In this work, we investigate two complementary techniques to improve datapath energy efficiency: *multi-level scheduling* and *hierarchical register files*. *Multi-level scheduling* partitions threads into two classes: (1) *active* threads that are issuing instructions or waiting on relatively short latency operations, and (2) *pending* threads that are waiting on long memory latencies. The cycle-by-cycle instruction scheduler only needs to consider the smaller pool of active threads, enabling a simpler and more energy-efficient scheduler. Our results show that a factor of 4 fewer threads can be active without suffering a performance penalty.

Hierarchical Register Files replace the monolithic register file with a multi-level register file hierarchy. Each level of the hierarchy has increasing capacity and corresponding increasing access energy. Our analysis of both graphics and compute workloads on GPUs indicates substantial register locality that can be exploited by storing short-lived values in low energy structures close to the execution units. We evaluate a range of design points including how many levels the hierarchy should contain, the size of each level, and how data movement should be controlled across the register file hierarchy. Each of these design points makes different tradeoffs in register file energy savings, storage requirements, and hardware and software changes required.

We explore both a hardware-managed register file cache and a software-managed operand register file. The hardware-managed register file cache is less intrusive on the software system, requiring only liveness hints to optimize the writeback of dead values.

However, this design requires tracking register file cache tags and modifications to the execution pipeline to add a register file tag check stage. A software-managed operand register file simplifies the microarchitecture by eliminating register file cache tags, but requires additional compiler support. The compiler must dictate, for each operand, which level of the hierarchy from which it should be accessed. Additionally, the compiler encodes in the instruction stream which instructions cause an active thread to be descheduled and must consider the two-level scheduler when allocating values to the register file hierarchy. The software-managed design is more efficient than the hardware-managed design because the compiler can use its knowledge of the data reuse patterns when making allocation decisions.

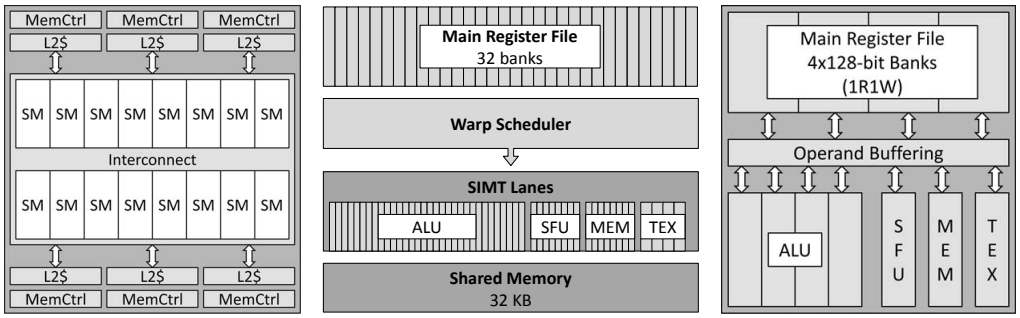
We reduce the storage requirements of the upper levels of our register file hierarchy by leveraging the multi-level scheduler to only allocate entries for active threads. This reduces the storage requirements by a factor of 4 but requires that values be preserved, in the main register file, when active threads are suspended. Our hardware-managed hierarchy writes back live values when a thread is suspended. With the software-managed hierarchy, the compiler ensures that values needed after a thread is descheduled are written to the main register file when the values are produced.

Our most efficient software-managed register file hierarchy reduces register file energy by 54%. This reduction corresponds to a chip wide savings of 5.5% of dynamic power. This savings is significant, as no single magic bullet will solve the energy efficiency problem. In today's highly competitive marketplace, a 5% energy savings can be crucial to a product's success. Future efficient systems will be achieved through a combination of many different optimizations such as the mechanisms proposed in this paper.

The remainder of this paper is organized as follows. Section 2 provides background on the design of contemporary GPUs and characterizes register value reuse in compute and graphics workloads. Section 3 describes our proposed two-level thread scheduler. Section 4 describes the microarchitecture of our register file hierarchy. Section 5 describes the compiler algorithms that are used to allocate values to our software controlled register file hierarchy. Section 6 describes our evaluation methodology. Section 7 presents performance and power results. Section 8 presents a limit study on possible further reductions in register file energy. Sections 9 and 10 discuss related work and conclusions.

2. BACKGROUND

While GPUs are becoming increasingly popular targets for computationally-intensive non-graphics workloads, their design is primarily influenced by triangle-based raster graphics. Graphics workloads have a large amount of inherent parallelism that can be easily exploited by a parallel machine. These workloads are characterized by having little control flow and high computational requirements. Most memory operations are unpredictable accesses to large texture arrays. These texture memory accesses tend to be fine-grained and difficult to prefetch. Texture working sets tend to be large (orders of magnitude larger than on-chip caches) and persist with temporal re-use occurring both within a single frame and between frames at long intervals in time. These streaming patterns are difficult to capture with traditional caches, which are targeted at reducing latency for smaller numbers of frequently accessed values. Texture caches typically cannot capture enough reuse across a thread's multiple texture accesses to consistently reduce thread execution latency, and individual thread execution latencies matter less than overall bandwidth. Texture caches are designed instead to conserve bandwidth by capturing spatial locality as well as temporal locality between neighboring pixels, which often access overlapping texture regions [Fatahalian and Houston 2008]. Be-



(a) Full chip (b) Streaming multiprocessor (SM) (c) 4-Wide SIMT lane detail

Fig. 1. Contemporary GPU architecture.

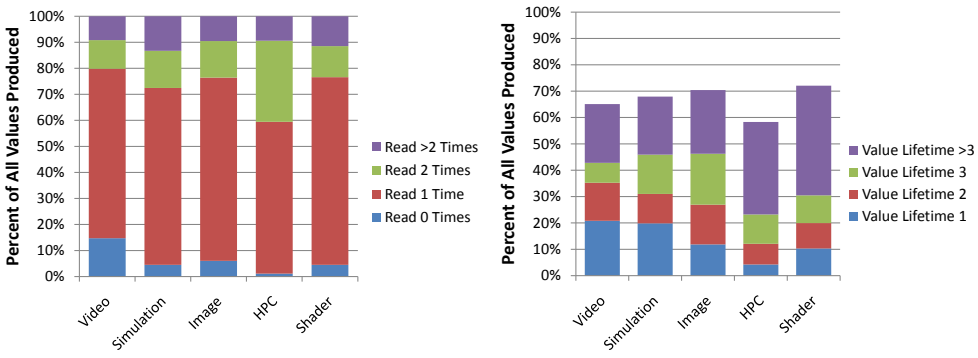
cause texture accesses are macroscopically unpredictable, and frequent, GPUs rely on massive multithreading to keep arithmetic units utilized.

Figure 1 illustrates the architecture of a contemporary GPU, similar in nature to NVIDIA's Fermi design. The figure represents a generic design point similar to those discussed in the literature [Bakhoda et al. 2009; NVIDIA 2009; Wong et al. 2010], but is not intended to correspond directly to any existing commercial product. The GPU consists of 16 streaming multiprocessors, 6 high-bandwidth DRAM channels, and an on-chip level-2 cache. A streaming multiprocessor (SM), shown in Figure 1(b), contains 32 SIMT (single-instruction, multiple thread) lanes that can collectively issue up to 32 instructions per cycle, one from each of 32 threads. Threads are organized into execution groups called *warps,* which execute together using a common physical program counter. While each thread has its own logical program counter and the hardware supports control-flow divergence of threads within a warp, the streaming multiprocessor executes most efficiently when all threads within a warp execute along a common control-flow path.

Fermi supports 48 active warps for a total of 1,536 active threads per SM. To accommodate this large set of threads, GPUs provide vast on-chip register file resources. Fermi provides 128KB of register file storage per streaming multiprocessor, allowing an average of 21 registers per thread at full scheduler occupancy. The total register file capacity across the chip is 2MB, substantially exceeding the size of the L2 cache. GPUs from AMD traditionally have even larger register files. The latest GPU from AMD contains 8.25MB of register file per chip [Smith 2011]. GPUs rely on heavily banked register files in order to provide high bandwidth with simple register file designs [NVIDIA 2008; Wong et al. 2010]. Despite aggressive banking, these large register files not only consume area and static power, but result in high per-access energy due to their size and physical distance from execution units. Prior work examining a previous generation NVIDIA GTX280 GPU (which has 64 KB of register file storage per SM), estimates that nearly 10% of total GPU power is consumed by the register file [Hong and Kim 2010]. Our own estimates show that the access and wire energy required to read an instruction's operands is twice that of actually performing a fused multiply-add [Galal and Horowitz 2011]. Because power-supply voltage scaling has effectively come to an end, driving down per-instruction energy overheads will be the primary way to improve future processor performance [ITRS 2009].

2.1. Baseline SM Architecture

In this work, we focus on the design of the SM (Figures 1(b) and 1(c)). For our baseline, we model a contemporary GPU streaming multiprocessor with 32 SIMT lanes.



(a) Number of reads per register value. (b) Lifetime of values that are read only once.

Fig. 2. Value usage characterization of GPU traces.

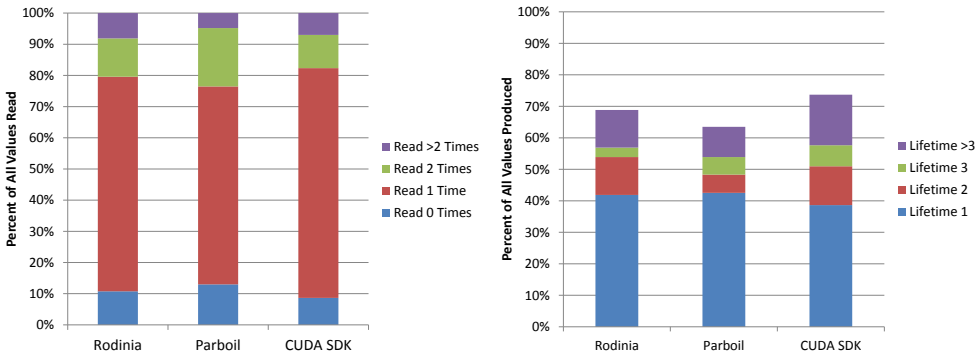
Our baseline architecture supports a 32-entry warp scheduler, for a maximum of 1024 threads per SM, with a warp issuing a single instruction across the 32 lanes per cycle. We model single-issue, in-order pipelines for each lane. Each SM provides 32KB of local scratch storage known as *shared memory*. Figure 1(c) provides a more detailed microarchitectural illustration of a cluster of 4 SIMT lanes. A cluster is composed of 4 ALUs, 4 register banks, a special function unit (SFU), a memory unit (MEM), and a texture unit (TEX) shared between two clusters. Eight clusters form a complete 32-wide SM.

A single-precision fused multiply-add requires three register inputs and one register output per thread for a total register file bandwidth of 96 32-bit reads and 32 32-bit writes per cycle per SM. The SM achieves this bandwidth by subdividing the register file into multiple dual-ported banks (1 read and 1 write per cycle). Each entry in the SM's main register file (MRF) is 128 bits wide, with 32 bits allocated to the same-named register for threads in each of the 4 SIMT lanes in the cluster. Each bank contains 256 128-bit registers for a total of 4KB. The MRF consists of 32 banks for a total of 128KB per SM, allowing for an average of 32 registers per thread, more than Fermi's 21 per thread. The trend over the last several generations of GPUs has been to provision more registers per thread, and our workloads make use of this larger register set.

The 128-bit registers are interleaved across the register file banks to increase the likelihood that all of the operands for an instruction can be fetched simultaneously. Instructions that require more than one register operand from the same bank perform their reads over multiple cycles, eliminating the possibility of a stall due to a bank conflict for a single instruction. Bank conflicts from instructions in different warps may occur when registers map to the same bank. Our MRF design is over-provisioned in bandwidth to reduce the impact of these rare conflicts. Bank conflicts can also be reduced significantly via the compiler [Zhuang and Pande 2003]. The operand buffering shown between the MRF and the execution units represents interconnect and pipeline storage for operands that may be fetched from the MRF on different cycles.

2.2. GPU Value Usage Characterization

Prior work in the context of CPUs has shown that a large fraction of register values are consumed a small number of times, often within a few instructions of being produced [Franklin and Sohi 1992]. Our analysis of GPU workloads indicates that these same trends hold. Figures 2(a) and 3(a) show the number of times a value written to a register is read for a set of real-world graphics and compute workloads. Details



(a) Number of reads per register value. (b) Lifetime of values that are read only once.

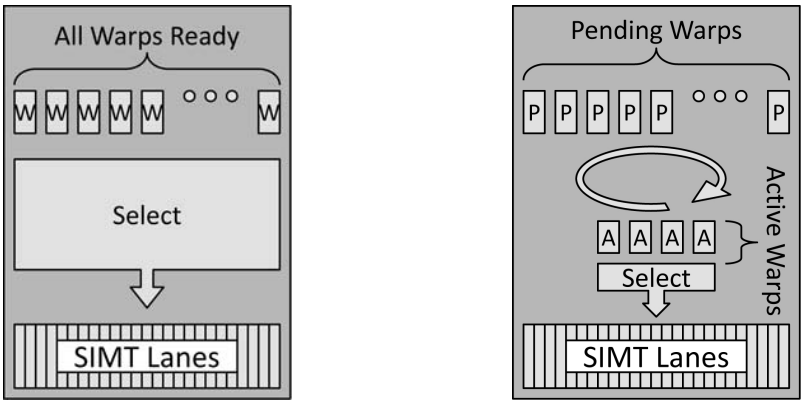
Fig. 3. Value usage characterization of CUDA applications.

on these benchmarks are presented in Section 6.1. Up to 70% of values are read only once, and only 10% of values are read more than twice. HPC workloads show the highest level of reuse with 40% of values being read more than once. Graphics workloads, labeled *Shader*, show reuse characteristics similar to the remaining compute traces. Figures 2(b) and 3(b) shows the lifetime of all dynamic values that are read only once. Lifetime is defined as the number of instructions between the producer and consumer (inclusive) in a thread. A value that is consumed directly after being produced has a lifetime of 1. Up to 40% of all dynamic values are read only once and are read within 3 instructions of being produced. In general, the HPC traces exhibit longer lifetimes than the other compute traces, due in part to hand-scheduled optimizations in several HPC codes where producers are hoisted significantly above consumers for improved memory level parallelism. Graphics traces also exhibit a larger proportion of values with longer lifetimes, especially compared to the CUDA workloads, due to texture instructions, which the compiler hoists to improve performance. These value usage characteristics motivate the deployment of a register file hierarchy to capture short-lived values and dramatically reduce accesses to the main register file.

3. TWO-LEVEL WARP SCHEDULER

The warp scheduler, shown in Figure 4(a), is responsible for keeping the SIMT cores supplied with work in the face of both pipeline and memory latencies. To hide long latencies, GPUs allocate a large number of hardware thread contexts for each set of SIMT cores. This large set of concurrently executing warps in turn increases scheduler complexity, thus increasing area and power requirements. Significant state must be maintained for each warp in the scheduler, including buffered instructions for each warp. In addition, performing scheduling among such a large set of candidate warps requires complex selection logic and policies.

The scheduler attempts to hide two distinct sources of latency in the system: (1) long, often unpredictable latencies, such as loads from DRAM or texture operations; and (2) shorter, often fixed or bounded latencies due to ALU operations, branch resolution, or accesses to the SM's local shared memory. The short-latency events are between 8 and 20 cycles, while the long-latency events require 400 cycles to complete. A large pool of available warps is required to tolerate latencies in the first group, but a much smaller pool of warps is sufficient to tolerate common short latencies. The latency of arithmetic operations and shared memory accesses along with the amount of per-thread ILP influences the number of threads required to saturate the hardware. Reducing the set of warps available for selection on a given cycle can reduce both the complexity and



(a) Single-level (b) Two-level

Fig. 4. Warp schedulers.

energy overhead of the scheduler. One important consequence of reducing the number of concurrently active threads is that it reduces the immediate-term working set of registers.

We propose a *two-level warp scheduler* that partitions warps into an *active* set eligible for execution and an inactive *pending* set. The smaller set of active warps hides common short latencies, while the larger pool of pending warps is maintained to hide long-latency operations and provide fast thread switching. Figure 4 illustrates a traditional single-level warp scheduler and our proposed two-level warp scheduler. All hardware-resident warps have entries in the outer level of the scheduler and are allocated MRF entries. The outer scheduler contains a large set of entries where *pending* warps may wait on long-latency operations to complete, with the number of pending entries required primarily influenced by the memory latency to be hidden. The inner level contains a much smaller set of *active* warps available for selection each cycle and is sized such that it can cover shorter latencies due to ALU operations, branch resolution, shared memory accesses, or cache hits. When a warp encounters a stall-inducing event, that warp can be removed from the active set but left pending in the outer scheduler. Introducing a second level to the scheduler presents a variety of new scheduling considerations for selection and replacement of warps from the active set.

Scheduling: For a two-level scheduler, we consider two common scheduling techniques: round-robin and greedy. For round-robin, we select a new ready active warp from the active warp pool each cycle using a rotating priority. For greedy, we continue to issue instructions from a single active warp for as long as possible, without stalling, before selecting another ready warp. Our single-level scheduler has the same options, but all 32 warps remain selectable at all times. We evaluate the effectiveness of these policies in Section 7.

Replacement: A two-level scheduler must consider when to remove warps from the active set. Only warps which are ready or will be ready soon should be kept in the active set; otherwise, they should be replaced with ready warps to avoid stalls. Replacement can be done preemptively or reactively, and depending on the size of the active set and the latencies of key operations, different policies will be appropriate. We choose to suspend active warps when they consume a value produced by a long-latency operation. Instructions marked by the compiler as sourcing an operand produced by a long-latency operation induce the warp to be suspended to the outer scheduler. When a warp is suspended it is removed from the active set and placed in the pending set.

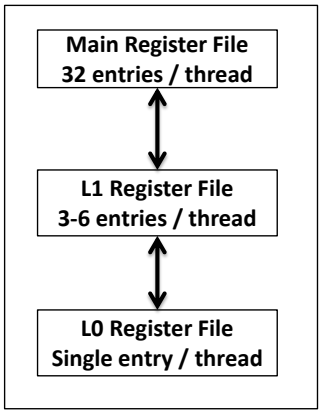


Fig. 5. High level view of our proposed register file hierarchy.

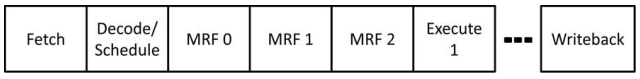
When the active set has a free entry, the scheduler moves the next ready warp from the pending set to the active set. We use round robin scheduling among warps in the pending set when choosing which warp to activate. We consider texture operations and global (cached) memory accesses as long-latency. This preemptive policy speculates that the value will not be ready immediately, a reasonable assumption on contemporary GPUs for both texture requests and loads that may access DRAM. Alternatively, a warp can be suspended after the number of cycles it is stalled exceeds some threshold; however, because long memory and texture latencies are common, this strategy reduces the effective size of the active warp set and sacrifices opportunities to execute instructions. For stalls on shorter latency computational operations or accesses to shared memory (local scratchpad), warps retain their active scheduler slot. For different design points, longer computational operations or shared memory accesses could be triggers for eviction from the active set.

4. REGISTER FILE HIERARCHY

In Section 2.2, we show that up to 40% of all dynamic register values are read only once and within 3 instructions of being produced. Because these values have such short lifetimes, writing them into the main register file wastes energy. We propose implementing a register file hierarchy so that these short-lived values can be captured in a low-energy storage structure. The upper levels of the register file hierarchy filter requests to the main register file (MRF) and provide several benefits: (1) reduced MRF energy by reducing MRF accesses; (2) reduced operand delivery energy, since the upper levels of the register file hierarchy can be physically closer to the ALUs than the MRF; and (3) reduced MRF bandwidth requirements, allowing for a more energy-efficient MRF. The majority of this work focuses on (1) while we discuss (2) and (3) in Section 7.8. The high level design of our register file hierarchy is shown in Figure 5. The upper level, L0 register file is the smallest and has the lowest access energy. The MRF in our hierarchy is identical to our baseline 128KB banked MRF. Values in the L0 and L1 register file always have a corresponding entry in the MRF. Therefore, if a value needs to be flushed to the MRF, no extra storage is required.

When designing the register file hierarchy we must address the following design parameters:

— Control: How data movement will be controlled across the register file hierarchy.
— Organization: How many levels will be present in the hierarchy and how many entries will be present in each level.



(a) Baseline pipeline.

(b) Pipeline with register file cache.

Fig. 6. GPU pipelines.

— Connectivity: How the execution units are connected to each level in the register file hierarchy.

4.1. Hardware-Managed Cache

First, we explore a design that uses hardware controlled caching to move values through the register file hierarchy. In this design, we augment the MRF with a register file cache (RFC). We evaluate designs where this cache consists of either 1 or 2 levels. Using a RFC requires slight modification to the pipeline as shown in Figure 6. Figure 6(a) shows a baseline pipeline with stages for fetch, decode/schedule, register file access (3 cycles to account for fetching and delivering multiple operands), one or more execute stages, and writeback. The pipeline with register file caching adds a stage to check the register file cache tags to determine if the operands are in the RFC. Operands not found in the RFC are fetched over multiple cycles from the MRF as before. Operands in the cache are fetched during the last stage allocated to MRF access. The RFC is multiported and all operands present in the RFC are read in a single cycle. We do not exploit the potential for reducing pipeline depth when all operands can be found in the RFC, as this optimization has a small effect on existing throughput-oriented architectures and workloads. The tag-check stage does not affect back-to-back instruction latencies, but adds a cycle to the branch resolution path. Our results show that this additional stage does not reduce performance noticeably, as branches do not dominate in the traces we evaluate.

RFC Allocation: Our baseline RFC design allocates the result of every operation into the upper level of the RFC. We explored an extension that additionally allocates RFC entries for an instruction's source operands. We found that this policy results in 5% fewer MRF reads with a large RFC, but also pollutes the RFC, resulting in 10-20% more MRF writes. Such a policy requires additional RFC write ports, an expense not justified by our results.

RFC Replacement: Prior work on register file caches in the context of CPUs has used either LRU replacement [Cruz et al. 2000] or a combination of FIFO and LRU replacement [Zeng and Ghose 2006] to determine which value to evict when writing a new value into the RFC. While our baseline RFC design uses a FIFO replacement policy, our results show that using LRU replacement results in only an additional 1-2% reduction in MRF accesses. Compared to prior work on CPU register file caching, our RFC can only accommodate very few entries per thread due to the large thread count of a throughput processor, reducing the effectiveness of LRU replacement.

RFC Eviction: In our hardware-managed design, a value that is evicted from the RFC must be written back to the MRF. Additionally, values evicted from the L0 RFC must be written back to the L1 RFC. However, many of the values evicted from the RFC are actually known statically to be dead at eviction time. In order to elide write-backs of these dead values, we consider a combined hardware/software RFC design. We extend our hardware-only RFC design with compile-time generated static liveness

information, which indicates the last instruction that will read a particular register instance. This information is passed to the hardware by an additional bit in the instruction encoding. Registers that have been read for the last time are marked dead in the RFC and their values need not be written back. This optimization is conservative and never destroys valid data that could be used in the future. Due to control flow uncertainty in the application, some values that are actually dead will be unnecessarily written back. This optimization reduces the number of writebacks to the MRF, but requires more information to be encoded in each instruction. We evaluate this tradeoff from an energy perspective in Section 7.

4.2. Software-Managed Operand Register File

In addition to a hardware-managed register file cache (RFC), we also explore an alternative design where the compiler controls all data movement through the register file hierarchy. In the hardware controlled RFC design, the compiler only provides liveness hints that are used to elide the writeback of dead values. The software scheme replaces the RFC with a compiler-managed operand register file (ORF). When a value is produced, the compiler must decide to which level of the hierarchy it should be written. This scheme has several advantages over the hardware-managed design. First, the compiler can leverage its knowledge of register usage patterns to determine where to allocate values across the register file hierarchy. The compiler-managed scheme eliminates all writebacks to the MRF. Rather than waiting for values to be evicted from the RFC, the compiler identifies values that must be written to the MRF when they are produced and encodes this information in the instruction. Removing the reads used for writebacks can save a substantial amount of energy. The compiler can also eliminate overhead writes to the RFC for values that will not be consumed soon—values which would get written to the RFC and evicted before they are read. Finally, using an ORF versus an RFC allows the location of a value to be determined when an instruction is decoded and removes the need to track tags for RFC entries.

A software-managed hierarchy requires compiler support to specify where values should be read from and written to. We discuss the allocation algorithms required in Section 5. The compiler analyzes each kernel and performs register allocation, attempting to maximize the number of accesses from the low-energy levels of the hierarchy and minimizing the number of accesses to the MRF. When a value is produced, it must be written to each level of the hierarchy from which it will later be consumed, eliminating the need for any writebacks. Each register specifier must indicate from which level of the hierarchy the value should be accessed. Rather than adding explicit extra bits to indicate the level, the register file namespace is partitioned, with each segment of architectural register names representing a different level of the hierarchy. The architecture namespace is under-utilized and using a portion of the namespace to represent ORF entries does not diminish a program's ability to fully utilize its MRF resources. Since the existing register file namespace is used, a binary can correctly execute on chips with different hierarchy organizations. However, modern NVIDIA GPUs rely on a runtime just-in-time (JIT) compiler which will have explicit knowledge of the register file hierarchy organization for the target system. We evaluate the encoding overheads in Sections 7.4 and 7.9.

4.3. Organization

We explored both a two-level and three-level register file hierarchy and find that a three-level hierarchy is more energy-efficient. In our designs, the L0 register file has a single 32-bit entry per thread. The L1 register file contains 3-6 entries per thread and the lower level is the MRF, containing 32 entries per thread, large enough to prevent spills for the majority of our GPU workloads. Our value usage analysis found many

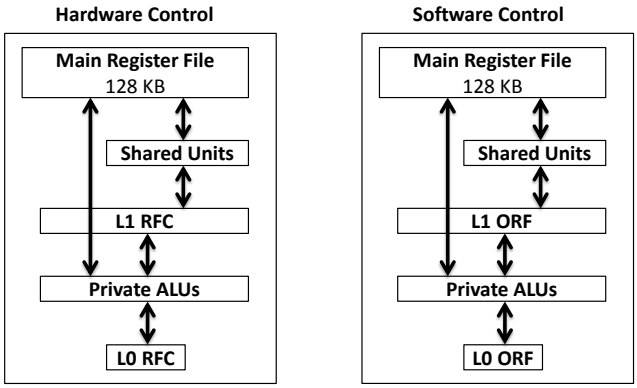


Fig. 7. Our proposed HW and SW register file hierarchies.

values whose only consumer was the next instruction. By introducing a single-entry upper level we are able to capture these values in a very low-energy structure, saving energy compared with a two-level design.

4.4. Connectivity

The ALUs operate with full warp wide throughput and we refer to them as the private datapath. The MEM, TEX, and SFU units, shown in Figure 1(c), operate at a reduced throughput and we collectively refer to them as the shared units. Only 7% of all values produced are consumed by the shared units. Further, 70% of the values consumed by the shared units are produced by the private datapath, reducing the opportunity to store values produced and consumed by the shared units near the shared units. Due to these access patterns, the most energy-efficient configuration is for the shared units to be able to access values from the L1 but not from the L0. By restricting the L0 to only be accessible from the private datapath, we minimize the ALU to L0 wire path and the energy required to traverse this path.

We also explore an alternative L0 ORF design that splits the L0 into separate banks for each operand slot. For example, a fused multiply-add (D = A * B + C) reads values from three register sources referred to as operands A, B, and C. In a split L0 design, rather than a single L0 bank per SIMT lane, each lane has a separate L0 bank for each of the three operand slots. Each of the three L0 banks per lane contains a single 32-bit entry. This split L0 design is only possible when using the compiler to manage data movement, as a caching scheme would not know to which bank a value should be written. The compiler encodes which L0 bank a value should be written to and read from. This design increases the effective size of the L0, while keeping the access energy minimal. It is uncommon for the short-lived values stored in the L0 to be consumed across different operand slots. When a value is consumed across separate operand slots, the compiler allocates it to the L1 ORF, rather than to the split L0. Using a split L0 design can increase wiring energy, a tradeoff we evaluate in Section 7.8.

4.5. Proposed Designs

Figure 7 shows the most efficient hardware and software controlled designs: a 3 level hierarchy, composed of an MRF and either a two-level RFC or a two-level ORF. Because the L0 RFC/ORF is designed to capture values that are produced and immediately consumed, we size it to only have a single entry per thread, minimizing the access energy. The L1 RFC/ORF is sized to have 3-6 entries per thread depending on the configuration; we explore this tradeoff in Section 7. To minimize the wiring distance

from the commonly used private ALUs, we do not allow the shared units to access the L0 RFC/ORF. Values that are read or written by the shared units must reside in the L1 RFC/ORF or the MRF. When using the software-managed scheme, the compiler ensures that values needed by the shared units are never allocated to the L0 ORF. A pure hardware-managed scheme requires full connectivity from all execution units to both the L0 and L1. Alternatively, the compiler can be used to insert an extra bit specifying that a result should not be cached in the L0, if it may be needed by the shared units. Our results assume this approach and we discuss the encoding overheads in Section 7. When we evaluate register file hierarchies with only two levels, we do not provision the L0 RFC/ORF and only have an L1 RFC/ORF and MRF which are all fully connected to the execution units.

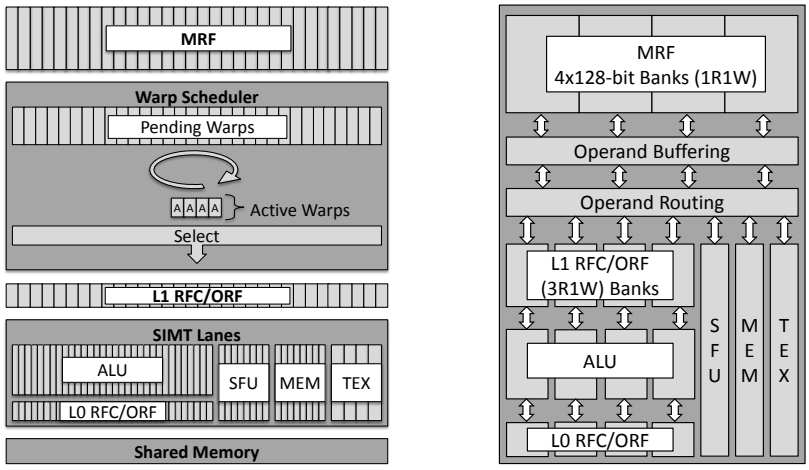
The MRF uses the same design described in Section 2, where the register file is composed of many SRAM banks that contain a single read and a single write port. Instructions that read multiple values from the MRF must do so over several cycles. The L0 and L1 RFC/ORF are built from flip-flop arrays which have 3 read ports and 1 write port. Having 3 read ports enables single-cycle operand reads, eliminating the costly operand distribution and buffering required for MRF accesses.

Our CUDA workloads use PTX (parallel thread execution) assembly code which supports 64-bit and 128-bit values stored across multiple 32-bit registers [NVIDIA 2011]. The only benchmark that makes significant use of double precision values is `dgemm`, where 90% of the instructions operate on double precision values. In all of the other benchmarks, fewer than 1% of the instructions operate on double precision values. When a larger width value is encountered, multiple entries are used to store the value in the register file hierarchy. This approach has the potential to increase the encoding overhead, which can be mitigated in future systems by restricting wide values to only occupy contiguous entries in the ORF. Due to the small number of wide values in our workloads, we make the simplifying assumption that values can be mapped to non-contiguous entries and ignore the encoding overhead. If applications become more double-precision heavy, the number of entries per-thread in the RFC/ORF may need to be increased to capture the larger working set of registers. Figure 26 shows that the software-managed register file hierarchy reduces register file energy by 56% on `dgemm`. Moving from a 3 entry per-thread ORF to a 7 entry per-thread ORF increases the savings in register file energy on `dgemm` from 56% to 60%. While `dgemm` benefits from a larger ORF, the presence of double precision values does not reduce the effectiveness of the register file hierarchy.

4.6. Minimizing Register File Storage Requirements

While two-level scheduling and a register file hierarchy are each beneficial in isolation, combining them substantially increases the opportunity for energy savings. Figure 8(a) shows our proposed architecture that takes advantage of a register file hierarchy to reduce accesses to the MRF while employing a two-level warp scheduler to reduce the required size of the upper levels of the register file hierarchy. Figure 8(b) shows the detailed SM microarchitecture which adds L0 and L1 RFC/ORF banks to our baseline GPU design.

To reduce the size of the RFC/ORF structures, entries are only allocated to *active* warps. When using hardware control, completed instructions write their results to the L0 RFC according to the policies discussed in Section 4.1. When a warp encounters a dependence on a long-latency operation, the two-level scheduler suspends the warp and evicts dirty RFC entries back to the MRF. To reduce writeback energy and avoid polluting the RFC, we augment the allocation policy described in Section 4.1 to bypass the results of long-latency operations around the RFC, directly to the MRF. Allocating entries only for active warps and flushing the RFC when an active warp is replaced



(a) High level SM architecture (b) Detailed microarchitecture

Fig. 8. Modified GPU microarchitecture. (a) High level SM architecture: MRF with 32 128-bit wide banks, multiported RFC/ORF (3R/1W per lane). (b) Detailed SM microarchitecture: 4-lane cluster replicated 8 times to form 32 wide machine.

increases the number of MRF accesses minimally but dramatically decreases the storage requirements of the RFC. When using software control, the compiler is responsible for deciding when threads will be deactivated due to long-latency operations. The compiler must ensure that any value needed after a thread is deactivated is present in the MRF, as the ORF is invalidated when an active thread is descheduled. Employing two-level scheduling allows us to reduce the storage requirements of the RFC/ORF by a factor of 4, since only active threads require RFC/ORF storage.

4.7. Instruction Encoding Overheads

Both the hardware and software managed schemes add one additional bit per-instruction to indicate that the next instruction may have a long-latency dependence. This bit triggers the active warp to be descheduled, allowing the long-latency operation to be resolved. In addition the various flavors of the hardware-managed schemes make the following encoding changes:

— Liveness hints: Along with the pure hardware-managed caching scheme, we evaluate using liveness hints from the compiler to prevent the writeback of dead values from the RFC to the MRF. This requires 1 extra bit for each source register to indicate that the given source register will never be read again. In the worse case, an instruction may require an additional 3 bits for operations such as a fused multiply-add (FMA). Section 7 evaluates the potential of these liveness hints to save energy by preventing the writeback of dead values. Our model in Section 6 estimates the upper bound for the additional fetch and decode energy of adding an additional bit to be 0.3% of chip-wide energy. System designers must balance the overheads of additional bits with the gains provided by these optimizations.
— L0 RFC caching: Because, our microarchitecture assumes that the shared units are not connected to the L0 RFC, values needed by the shared units must not be cached in the L0 RFC. Encoding this information requires adding one bit per instruction to indicate that the result may be needed by the shared units and should not be cached in the L0 RFC. Our results in Section 7 show that the gain in moving from a 2-level to a 3-level hardware-managed hierarchy is large and justifies this additional bit.

Along with the strand endpoint markers of 1 bit per instruction, the software-managed scheme requires the instruction encodings to allow each register operand to specify from which level of the hierarchy a value should be accessed. We propose to partition the register file namespace so that separate regions of the namespace correspond to different levels of the hierarchy. Our baseline approach allows a value to be written to 2 of the 3 levels of the register file hierarchy when it is produced. This flexibility puts pressure on the register file namespace. In Section 7 we explore restricting the ability to write a value to multiple levels of the hierarchy to simplify the encoding.

5. COMPILER ALLOCATION

In this section, we discuss the compiler support required for a software-managed register file hierarchy combined with a two-level warp scheduler. The compiler minimizes register file energy by both reducing the number of accesses to the MRF and by keeping values as close to the execution units that operate on them as possible. Allocating values to the various levels of our register file hierarchy is fundamentally different from traditional register allocation for several reasons. First, unlike traditional register allocation where a value's allocation location dictates access latency, in our hierarchy a value's allocation location dictates the access energy. The processor experiences no performance penalty for accessing values from the MRF versus the ORF. Second, because the ORF is temporally shared across threads, values are not persistent and must be stored in the MRF when warps are descheduled. The compiler controls when warps are descheduled, thus invalidating the ORF, forcing scheduling decisions to be considered when performing allocation. Rather than simply consider a value's lifetime, in order to maximize energy savings the compiler must consider the number of times a value is read and the location of these reads in relation to scheduling events. Finally, the allocation algorithms must consider the small size of the L0 ORF and to a lesser extent the L1 ORF, compared with traditional register allocation that has access to a larger number of register file entries. The compiler algorithms to share the register file hierarchy across threads in the most energy-efficient manner are a key contribution of this work.

5.1. Extensions to Two-Level Warp Scheduler

The two-level warp scheduler used with an RFC deschedules a warp when it encounters a dependence on a long-latency operation. These scheduling events can vary across executions due to control flow decisions. When using the two-level scheduler with the SW controlled ORF, the compiler must dictate when a warp is descheduled to prevent control flow from causing uncertainties in when a warp will be descheduled. The compiler performs ORF allocation for an execution unit called a *strand*. We define a strand as a sequence of instructions in which all dependences on long-latency instructions are from operations issued in a previous strand. This definition differs from prior work by Crago, on a decoupled access / execute design which defines strands as execution units with separate hardware contexts which either access or consume memory [Crago and Patel 2011]. Along with this prior work, we use the concept of strands to handle long-latency events. We add an extra bit to each instruction indicating whether or not the instruction ends a strand. We evaluate the energy overhead of adding this extra bit in Section 7.8. To simplify ORF allocation, we add the restriction that a backward branch ends a strand and that a new strand must begin for basic blocks that are targeted by a backwards branch. All values communicated between strands must go through the MRF. If a strand ends due to a dependence on a long-latency operation, the warp will be descheduled by the two-level warp scheduler until the long-latency operation completes. If a strand ends due to a backwards branch, the warp need not be descheduled. However, all inter-strand communication must always occur through the MRF.

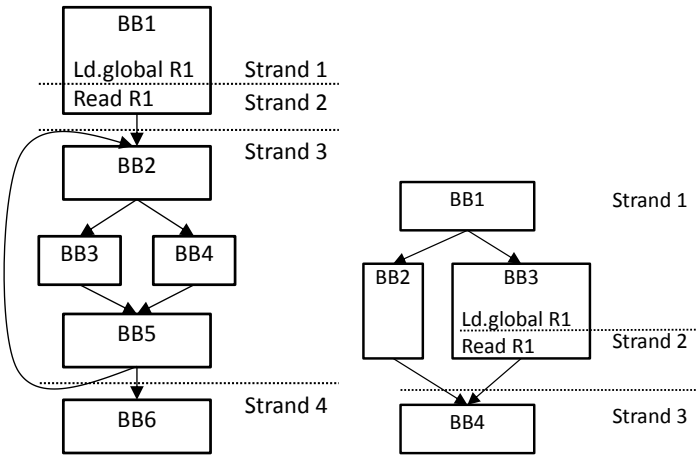


Fig. 9. Examples of strand endpoints.

Figure 9(a) shows the strand boundaries for a simple kernel. Strand 1 terminates due to a dependence on a long-latency operation, which will cause the warp to be descheduled. The other strand endpoints are due to the presence of a backwards branch. At these strand endpoints, the warp need not be descheduled, but values can not be communicated through the ORF past strand boundaries. Section 8 explores relaxing the requirement that strands may not contain backward branches. Figure 9(b) shows an example where a long-latency event may or may not be executed due to control flow. Uncertainty in the location of long-latency events complicates allocation; if `BB3` executes, the warp will be descheduled to resolve all long-latency events. In `BB4`, the compiler must know which long-latency events are pending to determine when the warp will be descheduled. We resolve the uncertainty by inserting an extra strand endpoint at the start of `BB4`, preventing values from being communicated through the ORF. Because this situation is rare, the effect on performance by extra strand endpoints is negligible.

5.2. Baseline Algorithm

We first discuss our allocation algorithm assuming a two-level register file hierarchy (single-level ORF and MRF) and that values in the ORF cannot cross basic block boundaries. The input to our allocation algorithm is PTX assembly code which has been scheduled and register allocated [NVIDIA 2011]. PTX code is in pseudo-SSA form, which lacks phi-nodes. First, we determine the strand boundaries, across which all communication must be through the MRF. Next, we calculate the energy savings of allocating each value to the ORF using the function in Figure 10. We calculate the number of reads in the strand for each value and if each value is live-out of the strand. Live-out values must be written to the MRF, since the ORF is invalidated across strand boundaries. These values may also be written to the ORF if the energy savings from the reads outweighs the energy overhead of writing to the ORF. Accounting for the number of reads allows us to optimize for values that are read several times, which save the most energy when allocated to the ORF.

Figure 11 shows our baseline greedy algorithm. For each strand, all values produced in that strand are sorted in decreasing order based on a weighted measure of the energy saved by allocating them to the ORF, divided by the number of instruction slots they would occupy the ORF. Scaling the energy savings by the length of time the ORF

```
savings = NumberOfReadsInStrand * (MRF_ReadEnergy - ORF_ReadEnergy) - ORF_WriteEnergy;
if LiveOutOfStrand == false then
    savings += MRF_WriteEnergy;
end
return savings;
```

Fig. 10. Function to calculate energy savings of allocating a register instance to the ORF.

```
foreach strand ∈ kernel do
    foreach registerInstance ∈ strand do
        range = registerInstance.LastReadInStrand - registerInstance.CreationInstruction;
        savings = calcEnergySavings(registerInstance) / range;
        if savings > 0 then
            priority_queue.insert(registerInstance);
        end
    end
    while priority_queue.size() > 0 do
        registerInstance = priority_queue.top();
        foreach orfEntry ∈ ORF do
            begin = registerInstance.CreationInstruction;
            end = registerInstance.LastReadInStrand;
            if orfEntry.available(begin, end) then
                orfEntry.allocate(registerInstance);
                exit inner for loop;
            end
        end
    end
end
```

Fig. 11. Algorithm for performing ORF allocation

will be occupied prevents long lived values from occupying an entry when it may be more profitable to allocate a series of short lived values. Our algorithm only allocates values to the ORF that save energy, using the parameters given in Section 6.3. We attempt to allocate each value to the ORF from the time it was created to the last read in the strand. If a value is not live out of the strand and we are able to allocate it to the ORF, the value never accesses the MRF. The compiler encodes, in each instruction, whether the value produced should be written to the ORF, MRF, or both and if the read operands should come from the ORF or the MRF. Next, we extend our baseline algorithm to capture additional register usage patterns commonly found in our benchmarks.

5.3. Partial Range Allocation

Figure 12(a) shows an example where `R1` is produced, read several times, and then not read again until much later. This value will likely not be allocated to the ORF, under our baseline algorithm because it has a long lifetime and we optimize for energy savings divided by a value's lifetime. To optimize for this pattern, we augmented our baseline algorithm to perform *partial range allocation*, allowing a subset of a value's reads to be handled by the ORF with the remaining reads coming from the MRF. Because this optimization requires a write to both the ORF and the MRF, the savings from reading some of the values from the ORF must outweigh the energy overhead of writing the value to the ORF. The partial range always begins when the value is produced and ends with a read of that value. We extend our baseline algorithm by

```
add R1, R2, 3
add R3, R1, 3
add R4, R1, 3
. . .
Independent
Instructions
. . .
add R9, R1, 3
```

(a) Partial Range

```
add R1, R0, 3
add R2, R0, 3
add R3, R0, 3
add R4, R0, 3
add R5, R0, 3
add R6, R0, 3
add R7, R0, 3
add R8, R0, 3
```

(b) Read Operand

Fig. 12. Examples of allocation optimizations.

```
savings = (NumberOfReadsInStrand - 1) * (MRF_ReadEnergy - ORF_ReadEnergy) - ORF_WriteEnergy;
return savings;
```

Fig. 13. Function to calculate energy savings of allocating a read operand to the ORF.

trying to allocate a partial range when we fail to allocate a value to the ORF. We iteratively shorten the value's range by reassigning the last read in the strand to target the MRF rather than the ORF. The ORF allocation range is repeatedly shortened until we either find a partial range that can be allocated to the ORF or until it does not become energetically profitable to allocate the partial range to the ORF. This optimization raises a question of how to order allocation of values to the ORF. If we allocate partial ranges after full ranges, the ORF may already have been allocated to a value that saves less energy. However, our greedy algorithm performs well and partial range allocation converts a significant number of MRF reads into ORF reads.

5.4. Read Operand Allocation

Figure 12(b) shows an example of a value, `R0`, that is read several times in a strand but not written. Our baseline algorithm would require all reads of `R0` to come from the MRF, since we only allocate values that are produced by the function units into the ORF. To optimize these accesses, we extend our baseline algorithm to implement *read operand allocation*. Figure 13 shows the calculation that determines the energy savings of allocating a read operand to the ORF. The energy savings of a read operand differs from the energy savings of a write operand, since the first read must come from the MRF and the write to the ORF is strictly an energy overhead. In the example shown in Figure 12(b), if the full range of `R0` is allocated to the ORF, it will be written to the ORF when the first `add` instruction executes and remain in the ORF until the final `add` instruction executes. We extend our baseline algorithm by calculating the potential energy savings for all read operands in a strand and inserting them into the same priority queue used for ORF allocation of write operands shown in Figure 11. We prioritize read operands the same as write operands using the potential energy savings divided by the number of instruction slots a value would occupy the ORF. We also allow partial ranges of read operands to be allocated.

5.5. Handling Forward Branches

We found it somewhat common for a value to be written on both sides of a hammock and consumed at the merge point. We extend our baseline algorithm to allow such values to remain in the ORF across forward branches. Figure 14 illustrates three dif-

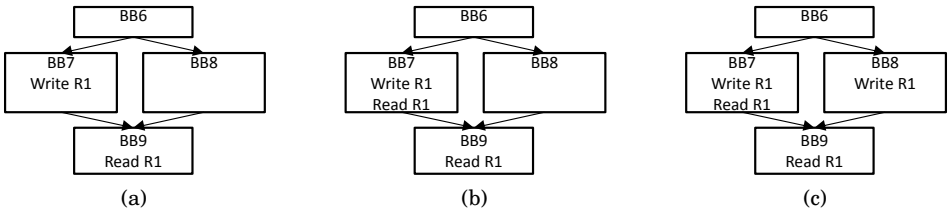


Fig. 14. Examples of impact of control flow on ORF allocation.

ferent patterns that occur with forward branches, assume that R1 has been written to the MRF by a previous strand. In Figure 14(a), R1 is written in BB7 and consumed in BB9. However, R1 cannot be allocated to the ORF because the read in BB9 must statically encode where R1 should be read from. Therefore, all accesses must be to the MRF. Alternatively, an extra move instruction could be inserted in BB8 to write R1 to the ORF, allowing the read in BB9 to come from the ORF. We do not allow for the insertion of these extra moves in our current design. In Figure 14(b), there is an additional read of R1 in BB7. Depending on the relative energy costs, it may be profitable to write R1 to both the ORF and MRF in BB7. The read in BB7 could then read R1 from the ORF, saving read energy. However, the read in BB9 must still read R1 from the MRF. In Figure 14(c), R1 is written in both BB7 and BB8. As long as both writes target the same ORF entry, the read in BB9 can be serviced by the ORF. Assuming R1 is dead after BB9, all accesses to R1 in Figure 14(c) are to the ORF, eliminating all MRF accesses. The compiler handles these three cases by ensuring that when a value's location is uncertain due to control flow, the value will always be available from the MRF.

5.6. Extending to Three-Level Hierarchy

We expand our baseline algorithm shown in Figure 11 to consider splitting values between the L0 ORF, L1 ORF, and MRF. When performing allocation, we first try to allocate as many values possible (both read and write operands) to the L0 ORF. Again, we prioritize values based on the energy savings of allocating them to the L0 ORF divided by the number of instruction slots they occupy the L0 ORF. Almost all values allocated to the L0 ORF have a lifetime of 1 or 2 instructions and are only read once. Next, we allocate as many of the remaining values as possible to the L1 ORF. Values that cannot be allocated to the L1 ORF are shortened using our previously discussed partial range allocation algorithm; we then attempt to allocate these shortened ranges to the L1 ORF. We explored allocating values to both the L0 ORF and L1 ORF, but found it rare to be energetically profitable. Therefore, in addition to the MRF, we allow a value to be written to either the L0 ORF or the L1 ORF but not both, simplifying the design. To minimize wire distance on the commonly traversed ALU to L0 ORF path, we restrict the L0 ORF to only be accessed from the private datapath. The compiler must ensure that values accessed by the shared units are only allocated into the L1 ORF. As discussed in Section 4.4, we explore using a split L0 ORF design, where each operand slot has a private L0 ORF bank. With this design, the compiler encodes which L0 ORF bank values should be written to and read from. Values that are accessed by more than one operand slot must be allocated to the L1 ORF.

6. METHODOLOGY

As described in Section 2.1, we model a contemporary GPU SIMT processor, similar in structure to the NVIDIA Fermi streaming multiprocessor (SM). Table I summarizes the simulation parameters used for our SM design. Standard integer ALU and single-precision floating-point operations have a latency of 8-cycles and operate with full

Table I. Simulation parameters.

Parameter	Value
Execution Model	In-order
Execution Width	32 wide SIMT
Register File Capacity	128 KB
Register Bank Capacity	4 KB
Shared Memory Capacity	32 KB
Shared Memory Bandwidth	32 bytes / cycle
SM External Memory Bandwidth	32 bytes / cycle
ALU Latency	8 cycles
Special Function Latency	20 cycles
Shared Memory Latency	20 cycles
Texture Instruction Latency	400 cycles
DRAM Latency	400 cycles

throughput across all lanes. While contemporary NVIDIA GPUs have longer pipeline latencies for standard operations [Wong et al. 2010], 8 cycles is a reasonable estimate based on AMD's GPUs [AMD 2010]. As with modern GPUs, various shared units operate with a throughput of less than the full SM SIMT width. Our texture unit has a throughput of four texture (`TEX`) instructions per cycle. Special operations, such as transcendental functions, operate with an aggregate throughput of eight operations per cycle.

Due to the memory access characteristics and programming style of the workloads we investigate, the system throughput is relatively insensitive to cache hit rates and typical DRAM access latency. Codes make heavy use of shared memory or texture for memory accesses, using most DRAM accesses to populate the local scratchpad memory. Combined with the large available hardware thread count, the relatively meager caches provided by modern GPUs only minimally alter performance results, especially for shader workloads. The performance difference between no caches and perfect caches is less than 10% for our workloads, so we model the memory system as bandwidth constrained with a fixed latency of 400 cycles.

6.1. Workloads

Table II shows our set of 210 real world gaming and compute instruction traces. The traces are encoded in NVIDIA's native ISA and taken from a variety of sources. Due to the large number of traces we evaluate, we present the majority of our results as category averages. Compute workloads, including high-performance and scientific computing, image and video processing, and simulation comprise 55 of the traces. The remaining 155 traces represent important shaders from 12 popular games published in the last 5 years. Shaders are short programs that perform programmable rendering operations, usually on a per-pixel or per-vertex basis, and operate across very large datasets with millions of threads per frame. We use these traces to evaluate our hardware-managed register file hierarchy. Because we must recompile applications to evaluate our software-managed designs, we only present results for the hardware-managed designs for the instruction traces.

To evaluate the software-managed register file hierarchy we use a collection of CUDA applications from the CUDA SDK [NVIDIA 2008], Rodinia [Che et al. 2009], MAGMA [MAGMA], and Parboil [PARBOIL] suites, shown in Table III. These benchmarks represent compute applications commonly run on modern GPUs. The CUDA SDK is released by NVIDIA and consist of a large number of kernels and applications designed to show developers the capabilities of modern GPUs. The Rodinia suite is designed for evaluating heterogeneous systems and is targeted to GPUs using CUDA or multicore CPUs using OpenMP. We use `dgemm` from the MAGMA library to high-

Table II. Trace characteristics.

Category	Examples	Traces	Avg. Dynamic Warp Insts.	Avg. Threads
Video Processing	H264 Encoder, Video Enhancement	19	60 million	99K
Simulation	Molecular Dynamics, Computational Graphics, Path Finding	11	691 million	415K
Image Processing	Image Blur, JPEG	7	49 million	329K
HPC	DGEMM, SGEMM, FFT	18	44 million	129K
Shader	12 Modern Video Games	155	5 million	13K

Table III. CUDA Benchmarks.

Suite	Benchmarks
CUDA SDK 3.2	BicubicTexture, BinomialOptions, BoxFilter, ConvolutionSeparable, ConvolutionTexture, Dct8x8, DwtHaar1D, Dxtc, EigenValues, FastWalshTransform, Histogram, ImageDenoising, Mandelbrot, MatrixMul, MergeSort, MonteCarlo, Nbody, RecursiveGaussian, Reduction, ScalarProd, SobelFilter, SobolQRNG, SortingNetworks, VectorAdd, VolumeRender
Parboil	cp, mri-fhd, mri-q, rpes, sad
Rodinia	backprop, hotspot, hwt, lu, needle, srad
MAGMA	dgemm

light the impact of double precision values. The Parboil suite is designed to exploit the massive parallelism available on GPUs; these applications generally have the longest execution times of all of our benchmarks. We present results for both the hardware-managed and software-managed designs for the CUDA benchmarks.

6.2. Simulation Methodology

To evaluate the performance impact of our two-level scheduler, we employ a custom trace-based simulator that models our modified SM pipeline. For the CUDA applications, we use Ocelot to generate dynamic execution traces from the PTX virtual instruction set representation of the program [Diamos et al. 2010]. When evaluating register file hierarchies, we simulate all threads in each trace. For two-level scheduling, we simulate execution time on a single SM for a subset of the total threads available for each workload, selected in proportion to occurrence in the overall workload. This strategy reduces simulation time while still accurately representing the behavior of the trace.

To evaluate our software controlled register file designs, we use Ocelot, an open source, dynamic compilation framework for PTX to run our static register allocation pass on each kernel [Diamos et al. 2010]. Ocelot provides useful compiler information such as dataflow analysis, control flow analysis, and dominance analysis. We augment Ocelot's internal representation of PTX to annotate each register access with the level of the hierarchy that the value should be read from or written to. Ocelot includes a PTX functional simulator, which we use to measure the number of accesses to each level of the register file hierarchy.

6.3. Energy Model

We model the energy requirements of several 3-read port, 1-write port RFC/ORF configurations using synthesized flip-flop arrays. We use Synopsys Design Compiler with both clock-gating and power optimizations enabled and commercial 40 nm high-performance standard cell libraries with a clock target of 1GHz at 0.9V. We estimate access energy by performing several thousand reads and writes of uniform random data across all ports. Table IV shows the RFC/ORF read and write energy for four 32-bit values, equivalent to one 128-bit MRF entry. We model the main register file (MRF) as a collection of 32 4KB, 128-bit wide dual-ported (1 read, 1 write) SRAM

Table IV. RFC/ORF area and read/write energy for 128-bit accesses.

RFC/ORF Entries per Thread	Active Warps								
	4			6			8		
	μm^2	Read (pJ)	Write (pJ)	μm^2	Read (pJ)	Write (pJ)	μm^2	Read (pJ)	Write (pJ)
4	5100	1.2	3.8	7400	1.2	4.4	9600	1.9	6.1
6	7400	1.2	4.4	10800	1.7	5.4	14300	2.2	6.7
8	9600	1.9	6.1	14300	2.2	6.7	18800	3.4	10.9

Table V. Modeling parameters.

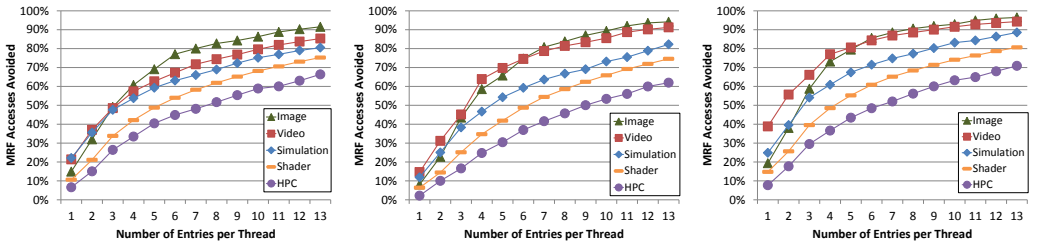
Parameter	Value
MRF Read/Write Energy	8/11 pJ
L0 ORF/RFC Read/Write Energy	0.7/2 pJ
MRF Bank Area	38000 μm^2
MRF Distance to Private	1 mm
L1 ORF Distance to Private	0.2mm
L0 ORF Distance to Private	0.05mm
MRF Distance to Shared	1 mm
L1 ORF Distance to Shared	0.4 mm
Wire capacitance	300 fF/mm
Voltage	0.9 Volts
Frequency	1 GHz
Wire Energy (32 bits)	1.9 pJ/mm

banks. SRAMs are generated using a commercial memory compiler and are characterized similarly to the RFC/ORF for read and write energy at 1GHz.

We model wire energy based on the methodology of [Kogge et al. 2008] using the parameters listed in Table V, resulting in energy consumption of 1.9pJ per mm for a 32-bit word. From a Fermi die photo, we estimate the area of a single SM to be 16 mm^2 and assume that operands must travel 1 mm from an MRF bank to the ALUs. Each RFC/ORF bank is private to a SIMT lane, greatly reducing distance from the RFC/ORF banks to the ALUs. The L0 RFC/ORF is only accessed by the ALUs in the private datapath, further reducing wiring energy. The L1 RFC/ORF has wire energy for the private datapaths 5 times lower than the MRF wire energy, while the L0 RFC/ORF has 20 times lower wire energy than the MRF. The tags for the RFC are located close to the scheduler to minimize the energy spent accessing them. Section 7.8 evaluates the impact of wire energy. Overall, we found our energy measurements to be consistent with previous studies [Balfour et al. 2009] and CACTI [Muralimanohar et al. 2009] after accounting for differences in design space and process technology.

To put our energy savings in context, we present a high-level GPU chip power model. A modern GPU chip consumes roughly 130 watts [Hong and Kim 2010]. The entire GPU board consumes 200-250 watts and includes the GPU chip, DRAM, cooling, and board level circuitry. This work seeks to optimize just the 130 watts consumed by the chip. If we assume that 1/3 of this power is spent on leakage, the chip consumes 87 watts of dynamic power. We assume that 30% of dynamic power is consumed by the memory system and 70% of dynamic power is consumed by the SMs [Leon et al. 2007]. This gives a dynamic power of 3.8 watts per SM for a chip with 16 SMs. The register file conservatively consumes 15% of the SM power, about 0.6 watts per SM [Hong and Kim 2010]. In Section 7 we evaluate our various proposals and their overall impact on chip-wide energy.

Several of our proposed designs require additional bits in the instruction encoding. To evaluate the impact on fetch and decode energy, we use the following model. Prior work has found that instruction fetch, decode, and schedule consumes roughly 15% of chip-wide dynamic power on a GPU. If we conservatively assume that fetch, decode,



(a) Read accesses avoided (b) Write accesses avoided (c) Write accesses avoided (SW liveness)

Fig. 15. Reduction of MRF accesses by baseline register file cache.

and schedule each consume roughly equal energy, fetch and decode are responsible for 10% of chip-wide dynamic power, which increases as bits are added to each instruction. To evaluate the impact of extra bits we make the simplifying assumption that additional bits result in linear increases in fetch and decode energy. Assuming the baseline system uses 32-bit words, each additional bit results in a 3% increase in fetch and decode energy. This corresponds to a chip-wide increase of 0.3% for each additional bit. We believe our estimates are conservative and the overhead would likely be smaller in a real system. Therefore, we use this metric as an upper bound for the increase in energy caused by the addition of extra bits in the instruction.

7. EVALUATION

This section demonstrates the effectiveness of register file hierarchies and two-level scheduling on GPU compute and graphics workloads. We first evaluate the effectiveness of each technique individually and then show how the combination reduces overall register file energy. As power consumption characteristics are specific to particular technology and implementation choices, we first present our results in a technology-independent metric (fraction of MRF reads and writes avoided), and then present energy estimates for our chosen design points.

7.1. Baseline Hardware Controlled Register File Cache

Using our GPU traces, first we evaluate a two-level hardware controlled register file hierarchy consisting of a single-level RFC and MRF, independent of the two-level scheduler and ignoring RFC flushes. Figures 15(a) and 15(b) show the percentage of MRF read and write traffic that can be avoided by the addition of the baseline single-level RFC. Even a single-entry RFC reduces MRF reads and writes, with the knee of the curve at about 6 entries for each per-thread RFC. At 6 RFC entries, this simple mechanism filters 45-75% of MRF reads and 35-75% of MRF writes. RFC effectiveness is lowest on HPC traces, where register values are reused more frequently and have longer average lifetimes, a result of hand scheduling.

As discussed in Section 2.2, many register values are only read a single time. Figure 15(c) shows the percentage of MRF writes avoided when static liveness information is used to identify the last consumer of a register value and avoid writing the value back to the MRF on eviction from the RFC. Read traffic does not change, as liveness information is used only to avoid writing back dead values. With 6 RFC entries per thread, the use of liveness information increases the fraction of MRF accesses avoided by 10–15%.

Figure 16 plots the reduction in MRF traffic with a 6-entry single-level RFC for each individual compute trace. For these graphs, each point on the x-axis represents a different trace from one of the sets of compute applications. The traces are sorted on the

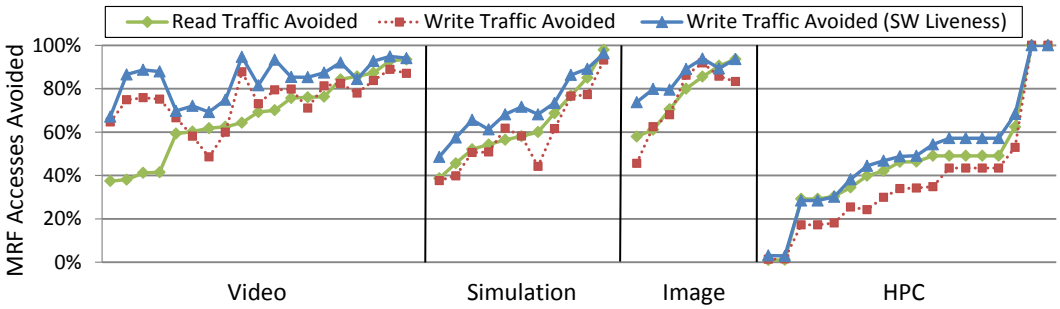
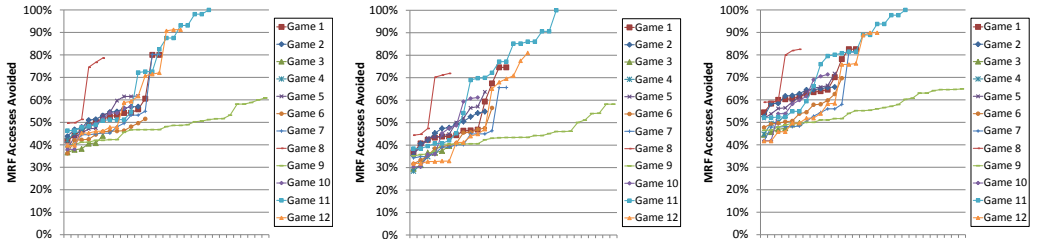


Fig. 16. Per-trace reduction in MRF accesses with a 6 entry RFC per thread (one point per trace).



(a) Read accesses avoided (b) Write accesses avoided (c) Write accesses avoided (with liveness info)

Fig. 17. Graphics per-trace reduction in MRF accesses with a 6 entry RFC per thread (one point per trace).

x-axis by the amount of read traffic avoided. The lines connecting the points serve only to clarify the data series and do not imply a parameterization of the series. The effectiveness of the RFC is a function of both the inherent data reuse in the algorithms and the compiler-generated schedule in the trace. Some optimizations such as hoisting improve performance at the expense of reducing the effectiveness of the RFC. All of the traces, except for a few hand-scheduled HPC codes, were scheduled by a production NVIDIA compiler that does not optimize for our proposed register file cache. The reduction in write traffic from using the compiler-provided liveness information is relatively constant across the traces. To provide insight into shader behavior, Figure 17 shows results for a 6-entry per thread single level RFC for individual shader traces grouped by games and sorted on the x-axis by MRF accesses avoided. Due to the large number of traces, individual datapoints are hard to observe, but the graphs demonstrate variability both within and across each game. Across all shaders, a minimum of 35% of reads and 30% of writes are avoided, illustrating the general effectiveness of this technique for these workloads. The use of liveness information to prevent the writeback of dead values removes 10-20% of the MRF writes. While the remainder of our register file hierarchy results for the traces are presented as averages, the same general trends appear for other RFC configurations.

7.2. Two-Level Warp Scheduler

Next, we consider the performance of our two-level warp scheduler rather than the typical, more complex, single level scheduler. Figure 18 shows SM instructions per clock (IPC) for a scheduler with 32 total warps and a range of active warps, denoted below each bar. Along with the arithmetic mean, the graph shows standard deviation across traces for each scheduler size. The scheduler uses a greedy policy in the inner level, issuing from a single warp until it can no longer issue without a stall, and uses

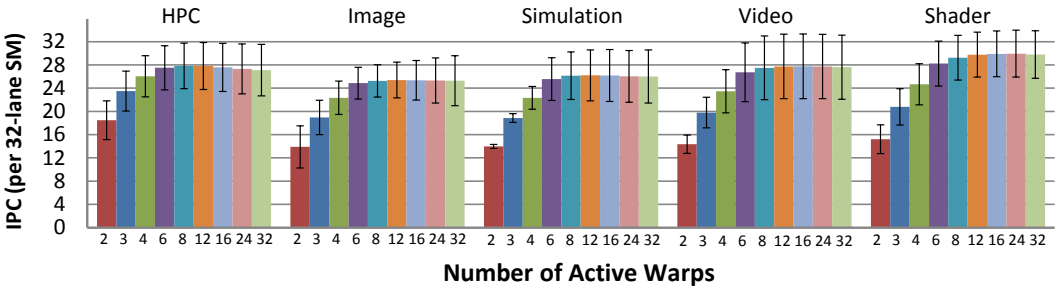
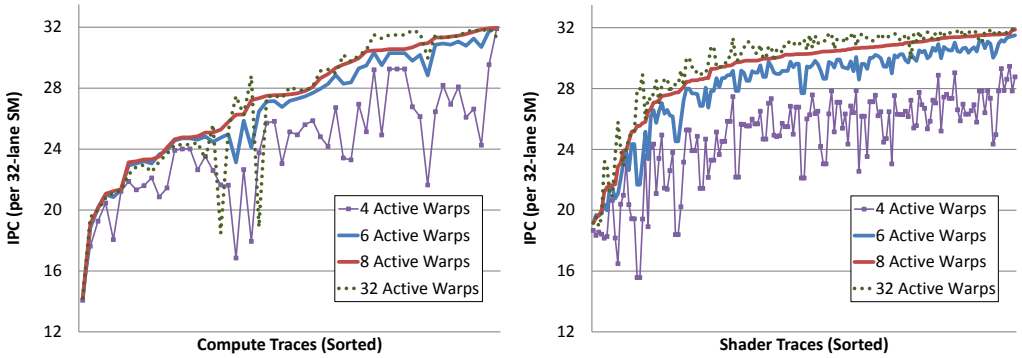


Fig. 18. Average IPC with +/-1 standard deviation for a range of active warps. Results for a single-level scheduler with all warps active are shown in bars labeled "32".



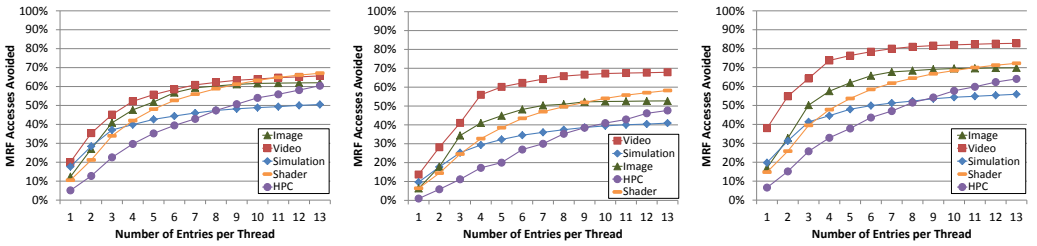
(a) 2-Level scheduling: compute (b) 2-Level scheduling: shader

Fig. 19. IPC for various active warp set sizes (one point per trace).

a round-robin policy when replacing active warps with ready pending warps from the outer level. The single-level scheduler (all 32 warps active) issues in the same greedy fashion as the inner level. A two-level scheduler with 8 active warps achieves nearly identical performance to a scheduler with all 32 warps active, while scheduling 6 active warps experiences a 1% performance loss on compute and a 5% loss on graphics shaders.

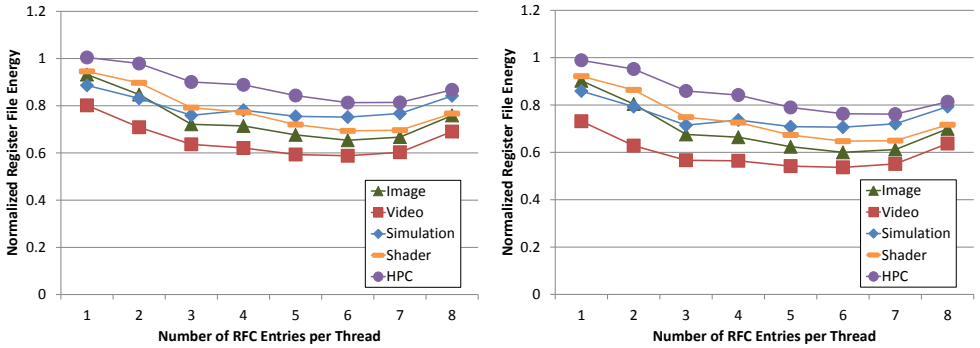
Figure 19 shows a breakdown of both compute and shader traces for a few key active scheduler sizes. The figure shows an all-warps-active scheduler along with three smaller active scheduler sizes. A system with 8 active warps achieves nearly the same performance as a single-level warp scheduler, whereas performance begins to deteriorate significantly with fewer than 6 active warps. The effectiveness of 6 to 8 active warps can be attributed in part to our pipeline parameters; an 8-cycle pipeline latency is completely hidden with 8 warps, while a modest amount of ILP allows 6 to perform nearly as well. Some traces actually see higher performance with fewer active warps when compared with a fully active warp scheduler; selecting among a smaller set of warps until a long-latency stall occurs helps to spread out long-latency memory or texture operations in time. We verify that the same trends hold for our CUDA applications.

For selection among active warps, we compared round-robin and greedy policies. Round-robin performs worse as the active thread-pool is increased beyond a certain size. This effect occurs because a fine-grained round-robin interleaving tends to expose long-latency operations across multiple warps in a short window of time, leading to many simultaneously stalled warps. For the SPMD code common to GPUs, round-



(a) MRF read accesses avoided (b) MRF write accesses avoided (c) MRF write accesses avoided (SW liveness)

Fig. 20. Effectiveness of single-level RFC when combined with two-level scheduler.



(a) Register file energy savings with baseline hardware scheme. (b) Register file energy savings with liveness hints.

Fig. 21. Energy savings for baseline hardware only scheme and compiler provided liveness hints.

robin scheduling of active warps leads to consuming easily extracted parallel math operations without overlapping memory accesses across warps. On the other hand, issuing greedily often allows a stall-inducing long-latency operation (memory or texture) in one warp to be discovered before switching to a new warp, overlapping the latency with other computation.

7.3. Combined Architecture

Next, we evaluate combing two-level scheduling with our baseline hardware-managed single level register file cache. A two-level scheduler dramatically reduces the size of the RFC by allocating entries only to active warps, while still maintaining performance comparable to a single-level scheduler. A consequence of two-level scheduling is that when a warp is deactivated, its entries in the RFC must be flushed to the MRF so that RFC resources can be reallocated to a newly activated warp.

Figure 20 shows the effectiveness of the single-level RFC in combination with a two-level scheduler on our GPU traces as a function of RFC entries per thread. Compared to the results shown in Figure 15, flushing the RFC entries for suspended warps increases the number of MRF accesses by roughly 10%. This reduction in RFC effectiveness is more than justified by the substantial (4×) reduction in RFC capacity requirements when allocating only for active warps.

7.4. Energy Savings of RFC

Next, we evaluate the energy savings of our two-level hardware controlled register file hierarchy on our set of GPU traces listed in Table II. Figure 21 shows register file

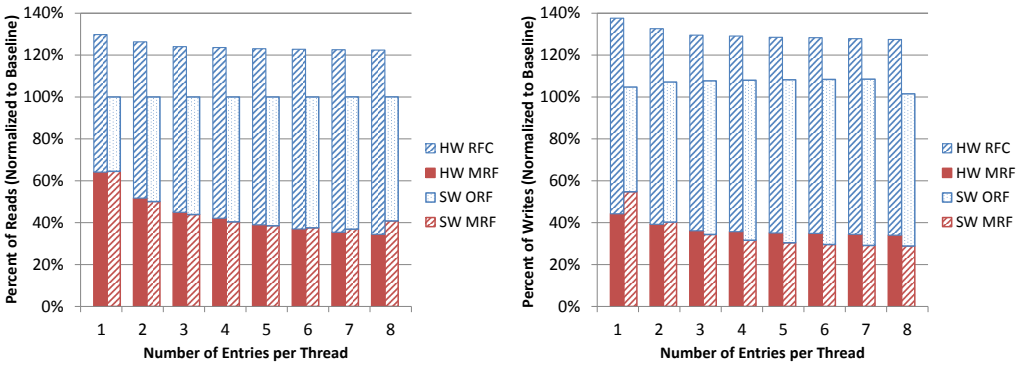
energy normalized to a baseline design that only contains a MRF. These results reflect the savings from both reducing access energy, from the RFC's smaller size, and reducing wire energy, as the RFC is located closer to the ALUs than the MRF. The x-axis shows the number of RFC entries per thread. A larger RFC filters more traffic from the MRF, but requires more energy to access. Generally, the most efficient design uses 6 RFC entries per thread. Figure 21(b) shows the energy savings when our baseline hardware design is augmented with compiler provided liveness information. This liveness information is used to prevent the writeback of dead values from the RFC to the MRF. The addition of liveness information provides a 10-20% reduction in register file energy. Corresponding with the results in Figure 20, the RFC is most effective for the Video processing traces and least effective for the HPC traces. However, a 6-entry RFC with liveness information provides at least a 20% reduction in register file energy across all categories of traces.

Based on our high-level GPU power model presented in Section 6, the most efficient hardware scheme with liveness hints results in a 5.4% reduction in SM dynamic energy and a 3.8% reduction in chip-wide dynamic energy. Passing the liveness hints from the compiler to the hardware requires 1 bit per source register. Instructions take between 0 and 3 source register operands. Our analysis in Section 6 estimates an upper-bound of 0.3% of chip-wide energy for each additional bit added to the instruction. This places an upper bound of 0.9% of chip-wide energy for passing the liveness information. The difference in register file energy with and without the liveness information is shown in Figure 21. System designers must carefully evaluate their designs to determine whether the register file energy reduction justifies the encoding overhead.

7.5. SW vs HW Control

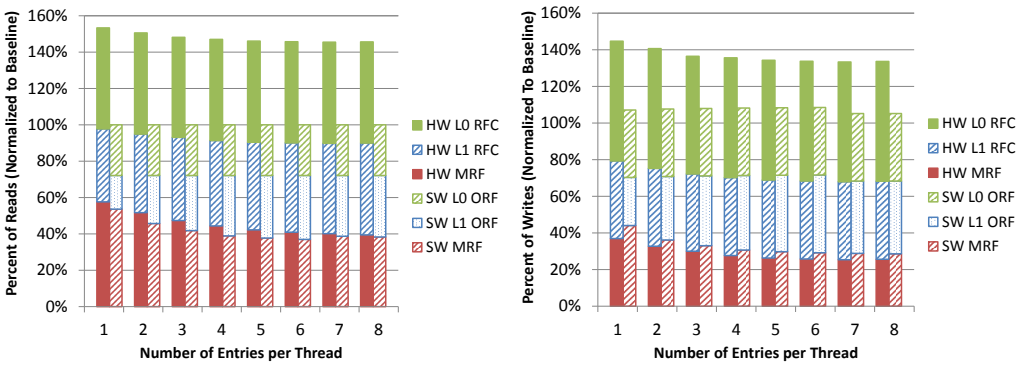
Next, we compare using a HW controlled RFC with our most basic SW controlled ORF using our selection of CUDA applications. Since we can not recompile the traces from Table II, the remainder of the results in this paper are from the CUDA applications listed in Table III. Figure 22 shows the breakdown of reads and writes across the hierarchies, normalized to the baseline system with a single-level register file. Compared with the software controlled ORF, the hardware controlled RFC performs 20% more reads, which are needed for writebacks to the MRF from the RFC. The compiler controlled scheme eliminates the writebacks by allocating each value to the levels of the hierarchy that it will be read from when it is produced. For probable ORF sizes of 2 to 5 entries per thread, the SW controlled scheme slightly reduces the number of reads from the MRF by making better use of the ORF. On average, each instruction reads 1.6 register operands and writes 0.8 register operands. Therefore, reductions in read traffic result in larger overall energy savings. The SW scheme reduces the number of writes to the ORF by roughly 20% compared to the RFC. Under the caching model, all instructions, except long-latency instructions, write their results to the RFC. However, some of these values will not be read out of the RFC. The compiler is able to allocate only the results that will actually be read from the ORF, minimizing unnecessary writes.

We have implemented several optimizations to our baseline SW scheme, including (1) partial range allocation, which allows a subset of a value's lifetime to be captured by the ORF, with the remaining reads serviced by the MRF; and (2) read operand allocation, which allocates read operands to the ORF along with write operands. These optimizations reduce the number of MRF reads by 20% at the cost of increasing ORF writes by 8%, Section 7.8 shows that this tradeoff saves energy.



(a) Read Breakdown (b) Write Breakdown

Fig. 22. Reads and writes to two-level register file hierarchy, normalized to single-level register file.



(a) Read Breakdown (b) Write Breakdown

Fig. 23. Reads and writes to three-level register file hierarchy, normalized to single-level register file.

7.6. Extending the Register File Hierarchy

Adding a small (1 entry / thread) L0 register file to the hierarchy has the potential for significant energy savings. We present results for both a software-managed and hardware-managed three-level hierarchy. When using a hardware-managed hierarchy, values produced by the execution units are first written into the L0 RFC. When a value is evicted from the L0 RFC, it is written back to the L1 RFC. Likewise when a value is evicted from the L1 RFC, it is written back to the MRF. As discussed in Section 4.5, the L0 RFC is only connected to the ALUs and not the shared units. Either the hardware must ensure that values needed by the shared units are not cached in the L0 RFC, or the shared units must be connected to the L0 RFC. We assume that the compiler uses an additional bit per instruction to specify values that should not be allocated in the L0 RFC. When a thread is descheduled, values in the RFC are written back to the MRF. We use static liveness information to inhibit the writeback of dead values. When using a software-managed hierarchy, the compiler controls all data movement across the three levels. Figure 23 shows the breakdown of reads and writes across the three levels of a software and hardware managed hierarchy. The software-managed design minimizes the number of overhead reads by eliminating writebacks. The software-managed design also reduces the number of MRF reads by making better use of the ORF. Despite the small size of the L0 ORF, it still captures 30% of all reads, resulting in

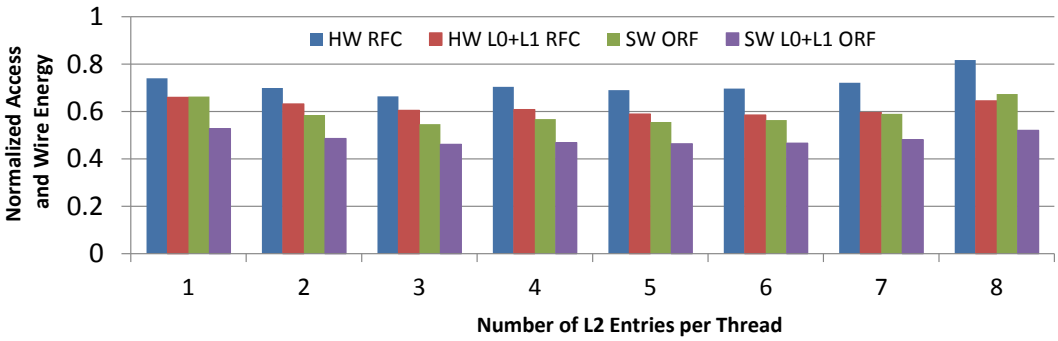


Fig. 24. Energy savings of register file organizations, normalized to baseline with single level register file.

substantial energy savings. Finally, the software-managed scheme reduces the number of overhead writes from 40% to less than 10% by making better allocation decisions. Control flow uncertainties slightly increase the number of MRF writes when using a software-managed design, presenting a minimal energy overhead.

7.7. Split vs Unified LRF

Finally, we consider the impact of a design with a separate L0 ORF bank for each operand slot. For example, a floating-point multiply-add (FMA) operation of D = A * B + C, reads operands from slots A, B, and C. Having a separate bank for each L0 ORF operand slot allows an instruction, in the best case, to read all of its operands from the L0 ORF. With a unified L0 ORF, only a single operand can be read from the L0 ORF and the others must be read from the L1 ORF. A split L0 ORF increases reads to the L0 ORF by nearly 20%, leading to decreases in both L1 ORF and MRF reads. Using a split L0 RF design has the potential to increase the wiring distance from the ALUs to the L0 ORF, a tradeoff we evaluate in Section 7.8.

7.8. Energy Savings

By combining the access counts from our various configurations with our energy model, we calculate the energy savings of various register file organizations. Figure 24 shows the register bank access and wire energy normalized to a single level register file hierarchy. These results assume 8 active warps to prevent a loss in performance.

Our results for the single-level HW controlled RFC, shown by the HW RFC bar, show a 34% savings in register file energy. By relying on software allocation and our optimizations to our baseline algorithm, we are able to improve this savings to 45%, as shown by the 3-entry per thread SW ORF bar. The software bars in Figure 24 include the partial range allocation and read operand optimizations to our baseline allocation algorithm, which provide a 3–4% improvement in energy efficiency. Allowing values to be live in the ORF past forward branches provides a 1.7% reduction in register file energy, compared to a design that restricts values allocated in the ORF to the life of a basic block. Compared to the HW RFC design, our optimized software system provides a 22% improvement in energy efficiency with no performance loss and a simplified microarchitecture that elides RFC tag storage and comparison. Comparing the most energy-efficient two-level designs, both the HW and SW schemes maximize the energy savings with 3 RFC/ORF entries per thread. The increase in effectiveness from having a larger RFC/ORF is not justified by the increased per-access energy.

Adding a L0 register file reduces energy for both the hardware and software managed schemes, although the benefit is larger when using software control, as shown by the HW L0+L1 RFC and SW L0+L1 ORF bars in Figure 24. The most energy-efficient

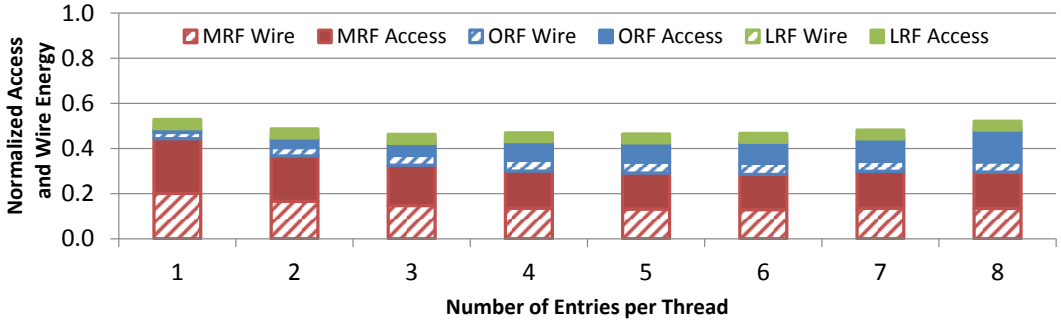


Fig. 25. Energy breakdown of most efficient design, normalized to baseline with single level register file.

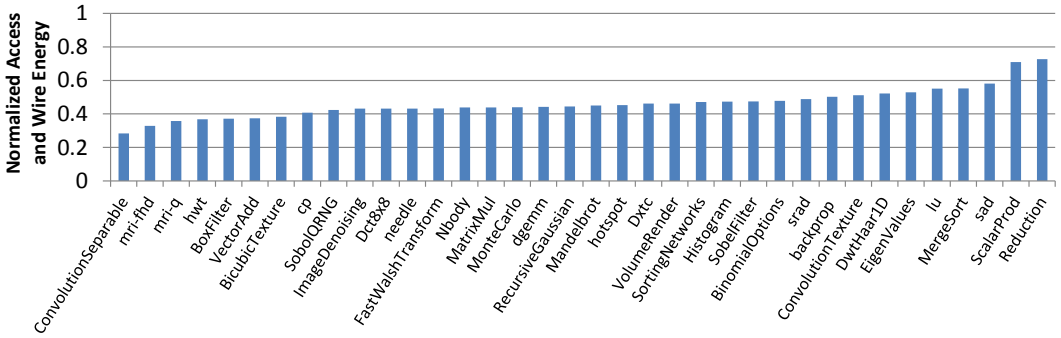


Fig. 26. Per benchmark access and wire energy results for the most energy efficient configuration, sorted by energy savings. (3-entry L1 ORF with split L0 ORF using read operand allocation and partial range allocation.)

three-level SW design uses 3 L1 ORF entries per thread, saving 54% of register file energy, while the most energy-efficient three-level HW design uses 6 L1 RFC entries per thread, saving 41% of register file energy. The compiler is better able to use each entry and the smaller structure reduces the per-access energy. Splitting the L0 ORF provides a 4% energy reduction compared with a unified L0 ORF. A split L0 ORF has the potential to increase L0 ORF wire distance and energy, which could negate the access energy savings.

Figure 25 shows the energy breakdown between wire and access energy among the different levels of the register file hierarchy for our most energy-efficient configuration. Roughly two thirds of the energy is spent on accesses to the MRF, equally split between access and wire energy. The bulk of the remaining energy is spent accessing values from the L1 ORF. Even though the L0 ORF captures $1/3$ of operand reads, accesses to the L0 ORF consume a very small portion of the overall energy, due to its small size, motivating future work to focus on reducing MRF accesses. Further, L0 ORF wire energy comprises less than 1% of the baseline register file energy.

Figure 26 shows the per-benchmark normalized register file energy for our most energy-efficient design, the SW L0+L1 ORF configuration with 3 L1 ORF entries per thread. `Reduction` and `ScalarProd` show the smallest efficiency gains of 25% and 30% respectively. These benchmarks see a small gain because they consist of a tight loop with global loads, a single floating-point multiply-add (FMA), and independent adds to update address variables and the loop counter. Few values are passed in registers between these instructions and the threads must be swapped in and out of the active set frequently due to the global loads, causing frequent invalidation of the ORF. The

best way to optimize these benchmarks is to unroll the inner loop and issue all of the long-latency instructions at the beginning of the loop. This strategy would allow the rest of the loop to remain resident and make use of the ORF.

Since we can not recompile the graphics shaders, we can not directly evaluate the register file energy savings from our software-managed design. However, these workloads have far fewer branches than the CUDA applications, which gives our software-managed design a larger window to optimize register allocation and simplifies allocation decisions. Therefore, we expect our software-managed register file hierarchy to provide significant energy savings on these graphics workloads. The data in Figure 2(b) shows that the Shader workloads generally have longer lifetimes compared to the compute traces and CUDA workloads. Therefore, they may benefit from slightly more ORF entries per thread than the compute traces and CUDA applications.

Our most energy-efficient three-level software control configuration saves 54% of register file energy, a 27% improvement over a purely HW controlled three-level design. Based on our high-level GPU power model presented in Section 6, our most efficient design reduces SM dynamic power by 8.3% and chip-wide dynamic power by 5.8%. Our system suffers no performance penalty and simplifies the microarchitecture of a hardware-managed design by eliminating register file cache tags.

7.9. Instruction Encoding Overhead

Our most efficient three level hardware-managed design makes the following changes to the instruction encoding:

— One bit per instruction to specify that an active warp should be descheduled after the current instruction.
— One bit per instruction to specify that values needed by the shared units should not be cached in the L0 RFC.
— One bit per source register operand to indicate that the register will not be read in the future. This liveness information is used to prevent the writeback of dead values to the MRF.

In the worst case, the hardware-managed design requires 5 bits per instruction. In Section 6 we estimate the upper bound for the extra fetch and decode energy to be 0.3% of chip-wide energy per additional bit. Therefore, the upper bound energy overhead for additional bit encodings for the three-level hardware design is 1.5%.

Our most efficient three level software-managed design makes the following changes to the instruction encoding:

— One bit per instruction to specify that an active warp should be descheduled after the current instruction.
— Register source and destination specifiers must be able to indicate the level of the hierarchy a value should be read from or written to.

We propose to partition the register namespace, with each level of the hierarchy having distinct ranges in the namespace. The approach works well with only slight pressure on the source register specifiers. One complication with our approach for the destination register specifiers is that we allow values to be written to up to two levels of the hierarchy. For example, a value can be written to both the L0 ORF and MRF or the L1 ORF and MRF. Since the L0 ORF is only a single entry, one bit of the register specifier can be used to control writes to the L0 ORF. Using a split LRF design increases the encoding pressure. Allowing arbitrary writes to any entry in the L1 ORF along with any entry in the MRF puts pressure on the namespace. Various restrictions can be placed on the encoding, such as requiring the L1 ORF name to be a function of the MRF name, for example the L1 ORF name could be computed as the MRF name

mod capacity of L1 ORF. These restrictions need to be considered in the context of a full system design.

We conduct two experiments to measure the usefulness of writing values to multiple levels of the hierarchy. First, we compare a design which only allows a value to be written to a single level with our design that allows a value to be written to up to two levels of the hierarchy. The more flexible design reduces register file energy by 7%, compared to the design that only allows a value to be written to a single level of the hierarchy. Alternatively, if we only allow writes to both the L0 ORF and MRF and disallow writes to both the L1 ORF and MRF this design comes within 2% of our more flexible design. Further, this design simplifies encoding, since writes to the L0 ORF can be controlled with a single bit in the destination register specifier. While encoding decisions must be carefully considered in the context of a full system design, our proposals for the software-managed design minimize the pressure on encoding space and incur a slight increase in overall chip power from fetch and decode overheads.

7.10. Additional Energy Savings Opportunities

In addition to the energy savings from simplifying the frequently traversed datapaths from operand storage to ALUs, the RFC/ORF and two-level scheduler provide two additional opportunities for energy savings. First, an RFC/ORF with 6 entries per thread reduces the average MRF bandwidth required per instruction by half. We expect that this effect can be leveraged to build a more energy-efficient MRF with less aggregate bandwidth, without sacrificing performance. Second, the two-level scheduler greatly simplifies scheduling logic relative to a complex all-warps-active scheduler, an energy-intensive component of the SM that must make time-critical selections among a large number of warps. By reducing the number of active warps, a two-level scheduler also reduces storage requirements for buffered instructions and scheduling state. Additionally, reduced ALU latencies can decrease the average short-term latency that the inner scheduler must hide, allowing fewer active warps to maintain throughput, further reducing RFC/ORF overhead. ALU latency of 4 cycles increases IPC for a 4-active scheduler by 5%, with smaller increases for 6 and 8. Finally, prior work on a previous generation GPU has shown that instruction fetch, decode, and scheduling consumes up to 15% of chip-level power, making the scheduler an attractive target for energy reduction [Hong and Kim 2010].

8. LIMIT STUDY OF CURRENT APPROACH

In this section, we consider several possible extensions to our work and the impact on energy efficiency of these extensions. The most energy-efficient of our designs, with a three-level software-managed hierarchy, reduces register file energy by 54%. An ideal system where every access is to the L0 ORF would reduce register file energy by 87%. This system is not practical for two reasons: the L0 ORF is too small to hold the working set of registers needed by our workloads and the L0 ORF is not preserved across strand boundaries. If every access were to a 5-entry per thread L1 ORF, register file energy would be reduced by 61%. In a realistic system, every level of the hierarchy will have to be accessed, with the MRF holding values needed across strand boundaries and the ORF holding temporary values. Our current system performs well and is competitive with an idealized system, where every access is to the L1 ORF.

8.1. Variable Allocation of ORF Resources

An alternative design would allow each strand to allocate a different number of ORF entries depending on its register usage patterns. We evaluate the potential of such a system by encoding in a strand header the energy savings of allocating between 1 and 8 ORF entries to each strand. When a strand is scheduled, the scheduler dynamically

assigns ORF resources based on the strand header and the other warps running in the system. If a thread receives fewer ORF entries than it expected, those values are serviced from the MRF as there is always a MRF entry reserved for each ORF value. We implement an oracle policy that examines the register usage patterns of future threads when deciding how many ORF entries to allocate. This variable allocation policy, using an oracle scheduler, is able to reduce register file energy by 6%. This policy also presents the opportunity to run with fewer active warps when sufficient ILP exists and to allocate each warp more ORF entries. If we optimistically assume that the number of warps can be lowered from 8 to an average of 6 across the program's execution, an additional 6% of register file energy can be saved.

While these idealized gains are enticing, there are several disadvantages to this dynamic policy. This policy requires the SM to track the dynamic mappings of active warps to ORF entries. Further, depending on the allocation policy, fragmentation within the ORF could occur. This approach requires the compiler to encode additional information in the program binary which takes energy to fetch and decode. We found that knowing the register needs of future threads was key in making allocation decisions. A realistic scheduler would perform worse than our oracle scheduler, unless restrictions were placed on thread scheduling to allow the scheduler insight into which threads would be scheduled in the future. Finally, running with fewer active threads has the potential to harm performance.

8.2. Allocating Past Backward Branches

We do not allow strands to contain backwards branches. Allocating values to the ORF for the life of a loop requires inserting explicit move instructions after the loop exits to move live values back to the MRF. Since we optimize for short-lived values and the ORF entries are already highly utilized resources, we expect that few values could be allocated to the ORF for the life of a loop. We examined the results of a variant of the hardware caching scheme and find the energy difference when allowing values to be resident in the cache past backward branches is only 5% over a system that flushes the RFC upon encountering a backward branch. We could expect to see similar or slightly better results using a SW controlled ORF, but the energy overhead of the explicit move instructions must be subtracted from the register file energy savings.

8.3. Instruction Scheduling

Reordering instructions presents two opportunities to reduce register file energy. The first is to reorder instructions within a basic block to shorten the distance between producers and consumers to increase ORF effectiveness. To evaluate the potential of this approach, we run our benchmarks with an 8-entry ORF but assume it has the same energy cost to access as a 3-entry ORF when making allocation decisions and calculating energy usage. This idealized configuration consumes 9% less energy than the realistic 3-entry ORF system. While this type of rescheduling has potential to increase the effective size of the ORF it is unlikely to increase it by nearly a factor of 3, as in our idealized experiment. A more realistic experiment increases the effective size of the ORF from 3 entries to 5 entries, which reduces register file energy by 6%.

The second potential for instruction scheduling is to move instructions across strand boundaries. Since inter-strand communication must go through the MRF, moving instructions across strand boundaries increases the number of values that are only accessed from the ORF. We calculate an idealized upper bound for moving instructions relative to long-latency events by never flushing the ORF when a warp is descheduled. In this idealized experiment, all machine resident warps, not just active warps, have ORF entries allocated, but we do not account for the higher access energy that these larger structures would require. This idealized system consumes 8% less energy

that the realistic system. Rescheduling would only be able to prevent a small number of values from being flushed, compared to our idealized experiment. Both of these scheduling techniques generally move consumers closer to producers, which has the potential to reduce performance. Given that the idealized energy gains are small, the realistic energy gains would be unlikely to justify any performance loss.

9. RELATED WORK

9.1. Tolerating Memory Latency

Our proposed two-level warp scheduler aims to minimize the size of the register file hierarchy and improve the energy efficiency of the scheduler while tolerating long-latency operations. Various past proposals have been explored to tolerate long-latency operations. MIT's Alewife used coarse-grained multithreading, performing thread context switches only when a thread incurs a cache miss and must access the network or on a failed synchronization attempt [Agarwal et al. 1990]. Tune proposed balanced multithreading, an extension to SMT where additional virtual thread contexts are presented to software to leverage memory-level parallelism, while fewer hardware thread contexts simplify the SMT pipeline implementation [Tune et al. 2004]. The Shared-Thread Multiprocessor proposed a shared pool of thread contexts that could be scheduled to any available core, allowing dynamic load balancing and fast context switching [Brown and Tullsen 2008]. Intel's Larabee proposed software mechanisms, similar to traditional software context switching, for suspending threads expected to become idle due to texture requests, which are long-latency operations [Seiler et al. 2008]. Mechanisms for efficient context switching have been proposed which recognize that only a subset of values are live across context switches [Nuth and Dally 1991]. We exploit this same phenomenon by only writing back the live values from the RFC to the main register file when an active threads is descheduled. Previous work on ILP-oriented superscalar schedulers has proposed holding instructions dependent on long-latency operations in separate waiting instruction buffers [Lebeck et al. 2002] for tolerating cache misses and using segmented [Raasch et al. 2002] or hierarchical [Brekelbaum et al. 2002] windows. Cyclone proposed a two-level scheduling queue for superscalar architectures to tolerate events of different latencies [Ernst et al. 2003].

9.2. Register File Architecture

The design of the register file system has always been important to minimize access time, area, and energy. Prior work has explored improving register files efficiency by reducing the number of entries [Ayala et al. 2003; Yu et al. 2011], reducing the number of ports [Park et al. 2002], and reducing the number of accesses [Park et al. 2006; Tseng and Asanovic 2000]. These approaches have been explored for traditional CPUs, VLIW processors [Zalamea et al. 2000; 2004; Zhang et al. 2005], and streaming processors [Dally et al. 2003; Rixner et al. 2000]. Prior work in the context of CPUs has found that a large number of register values are only used once and within a short period of time from when they are produced [Franklin and Sohi 1992]. Our analysis shows that these same trends hold for GPU workloads and we leverage these patterns when designing our GPU register file hierarchies.

As designers tried to increase clock rate and utilize heavily pipelined designs, the register file access time became a critical path, limiting cycle time. Several different approaches were explored to improve the register file cycle time, including using a hardware-managed register file cache [Cruz et al. 2000; Zeng and Ghose 2006; Jones et al. 2009; Nuth and Dally 1995; Borch et al. 2002; Balasubramonian et al. 2001]. Rather than reduce latency, our design for a register file cache aims to reduce the energy spent in the register file system. The challenges with GPU register caches are

different from CPU register caches because of the large number of threads and the throughput-oriented execution model of the GPU. CPU register caches have been proposed with tens of entries per threads [Cruz et al. 2000]. Due to the large number of threads present on a GPU, we explore designs with 3 to 6 entries per thread. Each thread on a GPU is executed in-order, removing several of the challenges faced by register file caching on a CPU, including preserving register values for precise exceptions [Hu and Martonosi 2000] and the interaction between register renaming and register file caching. Shioya et al. designed a register cache for a limited-thread CPU that aims to simplify the design of the MRF to save area and energy rather than reducing latency [Shioya et al. 2010]. Zeng and Ghose propose a register file cache that saves 38% of the register file access energy in a CPU by reducing the number of ports required in the main register file [Zeng and Ghose 2006]. We do not exploit the potential to redesign the main register file and focus instead on saving energy by reducing the number of accesses to the main register file. Similar to our use of liveness hints, past work has relied on software providing information to the hardware to increase the effectiveness of the register file cache [Jones et al. 2009].

9.3. Software Managed Register Hierarchy

Several systems, as early as the CRAY-1, have proposed a software-managed register file hierarchy to improve performance, energy, or area [Cooper and Harvey 1998; Russell 1978]. Swensen and Patt show that a 2-level compiler controlled register file hierarchy can provide nearly all of the performance benefit of a large register file on scientific codes [Swensen and Patt 1988]. ELM uses a software controlled two-level register file to save energy [Balfour et al. 2009]. Unlike our system, the upper level of the register file is not time-multiplexed across threads, allowing allocations to be persistent for extended periods of time. They use a similar algorithm for making allocation decisions that considers the number of reads and a value's lifetime, but do not directly use the energy savings when making allocation decisions. Follow on work considers the interaction between register allocation and instruction scheduling for ELM [Park and Dally 2011]. In Section 8 we explore allocating a variable number of ORF entries per thread. Using a variable number of registers, depending on a thread's usage, was proposed by [Yankelevsky and Polychronopoulos 2001]. Gebotys performs low-energy register and memory allocation by solving a minimum cost network flow [Gebotys 1997]. This approach requires a complicated algorithm for allocation and there is little to no opportunity to improve the allocations resulting from our greedy algorithm.

9.4. GPU Register File Design

Several different designs have been proposed to manage the large register files present on GPUs. AMD GPUs allocate short lived values to clause temporary registers [AMD 2011]. Unlike our RFC/ORF, these registers are managed by software and are not persistent across control flow boundaries. As Section 7.8 shows, a design that allows values to be live in the ORF past forward branches produces a meaningful register file energy reduction compared to a design that restricts values in the ORF to a basic block. In AMD GPUs the result of the last instruction can be accessed by a specially named register, similar to our L0 ORF. Unlike our L0 ORF, writes to this register cannot be inhibited so values can only be passed to the next instruction. Recent work proposes using a hybrid SRAM / embedded DRAM (eDRAM) register file to reduce the register file access energy of a GPU [Yu et al. 2011]. They modify the thread scheduler to minimize the impact of fetching values from eDRAM. As their proposed hybrid SRAM / eDRAM design is not yet a mature technology, the energy gains in a production system are unclear.

10. CONCLUSION

All modern computer systems are now power constrained. Future systems must be designed for energy efficiency as a primary constraint, as improvements in efficiency are the only road to improving performance. Over the last decade, increases in single thread performance have slowed, forcing system designers to exploit higher levels of parallelism to scale performance. Throughput oriented designs such as GPUs have gained popularity as a method of exploiting parallelism with a large number of simple, efficient cores. These designs leverage a massive level of multithreading to tolerate latency, necessitating very large register files and a complex thread scheduler. Future throughput processors will continue to scale both the number of processing elements and the number of threads per chip. The increase in thread count and the incrementally poor wire delay and energy scaling (relative to transistors) will make managing this large number of threads and their associated state more expensive.

We demonstrate that a combination of two design elements, multi-level scheduling and a register file hierarchy, can provide energy savings while maintaining performance for massively threaded GPU designs. Two-level scheduling reduces the hardware required for warp scheduling and enables a smaller, more efficient register file hierarchy. A register file hierarchy localizes accesses to small, energy-efficient structures near the execution units and filters accesses to the main register file. We evaluate a range of design points including how many levels the hierarchy should contain, the size of each level, and how data movement should be controlled across the register file hierarchy. Each of these design points makes different tradeoffs in register file energy savings, storage requirements, and hardware and software changes required.

The hardware-managed register file cache is least intrusive on the software system, requiring only liveness hints used to optimize the writeback of dead values. However, this design requires tracking register file cache tags and modification to the execution pipeline to add a register file tag check stage. Our most efficient hardware-managed design uses a three level hierarchy and reduces register file energy by 41%. A software-managed operand register file simplifies the microarchitecture by eliminating register file tags but requires additional compiler support. For each operand, the compiler must dictate from which level of the hierarchy it should be accessed. Additionally, the compiler encodes in the instruction stream which instructions end a strand and must consider the two-level scheduler when allocating values to the register file hierarchy. The added software complexity provides energy efficiency gains and our most efficient three level software-managed design reduces register file energy by 54%.

Our most efficient design results in a chip wide savings of 5.5% of dynamic power, a significant savings as there is no longer a single magic bullet to solve the efficiency problem. A 5% reduction in energy can be key to a product's success in today's highly competitive marketplace. In the past, designers relied on Dennard scaling to obtain nearly 3x improvement in energy efficiency per process generation. However, the effective end of voltage scaling now forces efficiencies to be wrung from the architecture. Current GPUs enjoy an advantage over CPUs in energy required per operation due to design decisions such as in-order execution and the SIMT execution model. However, substantial headroom in instruction energy-efficiency remains, as the energy to perform a basic computation is a small fraction of the overall instruction energy. Future energy efficient systems will be realized from the additive effects of many energy improvement techniques such as those demonstrated in this paper.

Acknowledgments

We thank the anonymous reviewers and the members of the NVIDIA Architecture Research Group for their comments. This research was funded in part by DARPA contract HR0011-10-9-0008 and NSF grant CCF-0936700.

REFERENCES

AGARWAL, A., LIM, B.-H., KRANZ, D., AND KUBIATOWICZ, J. 1990. APRIL: A Processor Architecture for Multiprocessing. In *International Symposium on Computer Architecture*. 104–114.

AMD. 2010. ATI Stream Computing OpenCL Programming Guide. `http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf`.

AMD. 2011. HD 6900 Series Instruction Set Architecture. `http://developer.amd.com/gpu/amdappsdk/assets/AMD_HD_6900_Series_Instruction_Set_Architecture.pdf`.

AYALA, J. L., VEIDENBAUM, A., AND LOPEZ-VALLEJO, M. 2003. Power-aware Compilation for Register File Energy Reduction. *International Journal of Parallel Programming 31,* 6, 451–467.

BAKHODA, A., YUAN, G. L., FUNG, W. W. L., WONG, H., AND AAMODT, T. M. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software*. 163–174.

BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. H. 2001. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *International Symposium on Microarchitecture*. 237–248.

BALFOUR, J., HARTING, R., AND DALLY, W. 2009. Operand Registers and Explicit Operand Forwarding. *IEEE Computer Archiecture Letters 8,* 2, 60–63.

BORCH, E., TUNE, E., MANNE, S., AND EMER, J. 2002. Loose Loops Sink Chips. In *International Symposium on High Performance Computer Architecture*. 299–310.

BREKELBAUM, E., RUPLEY, J., WILKERSON, C., AND BLACK, B. 2002. Hierarchical Scheduling Windows. In *International Symposium on Microarchitecture*. 27–36.

BROWN, J. A. AND TULLSEN, D. M. 2008. The Shared-thread Multiprocessor. In *International Conference on Supercomputing*. 73–82.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S. H., AND SKADRON, K. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Chaaracterization*. 44–54.

COOPER, K. D. AND HARVEY, T. J. 1998. Compiler-controlled Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2–11.

CRAGO, N. AND PATEL, S. 2011. OUTRIDER: Efficient Memory Latency Tolerance with Decoupled Strands. In *International Symposium on Computer Architecture*. 117–128.

CRUZ, J., GONZALEZ, A., VALERO, M., AND TOPHAM, N. P. 2000. Multiple-banked Register File Architectures. In *International Symposium on Computer Architecture*. 316–325.

DALLY, W. J., HANRAHAN, P., EREZ, M., KNIGHT, T. J., LABONTE, F., AHN, J.-H., JAYASENA, N., KAPASI, U. J., DAS, A., GUMMARAJU, J., AND BUCK, I. 2003. Merrimac: Supercomputing with Streams. In *International Conference for High Performance Computing*. 35–42.

DIAMOS, G., KERR, A., YALAMANCHILI, S., AND CLARK, N. 2010. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques*. 353 – 364.

ERNST, D., HAMEL, A., AND AUSTIN, T. 2003. Cyclone: A Broadcast-free Dynamic Instruction Scheduler with Selective Replay. In *International Symposium on Computer Architecture*. 253–263.

FATAHALIAN, K. AND HOUSTON, M. 2008. A Closer Look at GPUs. *Communications of the ACM 51,* 10, 50–57.

FRANKLIN, M. AND SOHI, G. S. 1992. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-grain Parallel Processors. In *International Symposium on Microarchitecture*. 236–245.

GALAL, S. AND HOROWITZ, M. 2011. Energy-Efficient Floating Point Unit Design. *IEEE Transactions on Computers 60,* 7, 913 – 922.

GEBOTYS, C. H. 1997. Low Energy Memory and Register Allocation Using Network Flow. In *Design Automation Conference*. 435–440.

HONG, S. AND KIM, H. 2010. An Integrated GPU Power and Performance Model. In *International Symposium on Computer Architecture*. 280–289.

HU, Z. AND MARTONOSI, M. 2000. Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics. In *Workshop on Complexity-Effective Design*.

ITRS 2009. International Technology Roadmap for Semiconductors. `http://itrs.net/links/2009ITRS/Home2009.htm`.

JONES, T. M., O'BOYLE, M. F. P., ABELLA, J., GONZÁLEZ, A., AND ERGIN, O. 2009. Energy-efficient Register Caching with Compiler Assistance. *ACM Transactions on Architecture and Code Optimization 6,* 4, 1–23.

KOGGE, P. ET AL. 2008. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Tech. Rep. TR-2008-13, University of Notre Dame. September.

LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. 2002. A Large, Fast Instruction Window for Tolerating Cache Misses. In *International Symposium on Computer Architecture*. 59–70.

LEON, A. S., LANGLEY, B., AND SHIN, J. L. 2007. The UltraSPARC T1 Processor: CMT Reliability. In *IEEE Custom Integrated Circuits Conference*. 555 –562.

MAGMA. MAGMA: Matrix Algebra for GPU and Multicore Architectures. `http://icl.eecs.utk.edu/magma`.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. P. 2009. CACTI 6.0: A Tool to Model Large Caches. Tech. rep., HP Laboratories. April.

NUTH, P. R. AND DALLY, W. J. 1991. A Mechanism for Efficient Context Switching. In *International Conference on Computer Design on VLSI in Computer & Processors*. 301–304.

NUTH, P. R. AND DALLY, W. J. 1995. The Named-State Register File: Implementation and Performance. In *International Symposium on High Performance Computer Architecture*. 4–13.

NVIDIA. 2008. Compute Unified Device Architecture Programming Guide Version 2.0. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`.

NVIDIA. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. `http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

NVIDIA. 2011. PTX: Parallel Thread Execution ISA Version 2.3. `http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/ptx_isa_2.3.pdf`.

PARBOIL. Parboil Benchmark Suite. `http://impact.crhc.illinois.edu/parboil.php`.

PARK, I., POWELL, M. D., AND VIJAYKUMAR, T. N. 2002. Reducing Register Ports for Higher Speed and Lower Energy. In *International Symposium on Microarchitecture*. 171–182.

PARK, J. AND DALLY, W. J. 2011. Guaranteeing Forward Progress of Unified Register Allocation and Instruction Scheduling. Tech. Rep. Concurrent VLSI Architecture Group Memo 127, Stanford University. March.

PARK, S., SHRIVASTAVA, A., DUTT, N., NICOLAU, A., PAEK, Y., AND EARLIE, E. 2006. Bypass Aware Instruction Scheduling for Register File Power Reduction. In *Languages, Compilers and Tools for Embedded Systems*. 173–181.

RAASCH, S. E., BINKERT, N. L., AND REINHARDT, S. K. 2002. A Scalable Instruction Queue Design Using Dependence Chains. In *International Symposium on Computer Architecture*. 318–329.

RIXNER, S., DALLY, W., KHAILANY, B., MATTSON, P., KAPASI, U., AND OWENS, J. 2000. Register Organization for Media Processing. In *International Symposium on High Performance Computer Architecture*. 375–386.

RUSSELL, R. M. 1978. The CRAY-1 Computer System. *Communications of the ACM 21,* 1, 63–72.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A Many-core x86 Architecture for Visual Computing. In *International Conference and Exhibition on Computer Graphics and Interactive Techniques*. 1–15.

SHIOYA, R., HORIO, K., GOSHIMA, M., AND SAKAI, S. 2010. Register Cache System not for Latency Reduction Purpose. In *International Symposium on Microarchitecture*. 301–312.

SMITH, R. 2011. AMD Radeon HD 7970 Review: 28nm And Graphics Core Next, Together As One. `www.anandtech.com/show/5261/amd-radeon-hd-7970-review`.

SWENSEN, J. A. AND PATT, Y. N. 1988. Hierarchical Registers for Scientific Computers. In *International Conference on Supercomputing*. 346–354.

TSENG, J. H. AND ASANOVIC, K. 2000. Energy-Efficient Register Access. In *Symposium on Integrated Circuits and Systems Design*. 377–382.

TUNE, E., KUMAR, R., TULLSEN, D. M., AND CALDER, B. 2004. Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy. In *International Symposium on Microarchitecture*. 183–194.

WONG, H., PAPADOPOULOU, M.-M., SADOOGHI-ALVANDI, M., AND MOSHOVOS, A. 2010. Demystifying GPU Microarchitecture through Microbenchmarking. In *International Symposium on Performance Analysis of Systems and Software*. 235–246.

YANKELEVSKY, M. N. AND POLYCHRONOPOULOS, C. D. 2001. α-coral: A Multigrain, Multithreaded Processor Architecture. In *International Conference on Supercomputing*. 358–367.

YU, W., HUANG, R., XU, S., WANG, S.-E., KAN, E., AND SUH, G. E. 2011. SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading. In *International Symposium on Computer Architecture*.

ZALAMEA, J., LLOSA, J., AYGUADE, E., AND VALERO, M. 2000. Two-level Hierarchical Register File Organization for VLIW Processors. In *International Symposium on Microarchitecture*. 137–146.

ZALAMEA, J., LLOSA, J., AYGUADE, E., AND VALERO, M. 2004. Software and Hardware Techniques to Optimize Register File Utilization in VLIW Architectures. *International Journal of Parallel Programming 32,* 6, 447–474.

ZENG, H. AND GHOSE, K. 2006. Register File Caching for Energy Efficiency. In *International Symposium on Low Power Electronics and Design*. 244–249.

ZHANG, Y., HE, H., AND SIN, Y. 2005. A New Register File Access Architecture for Software Pipelining in VLIW Processors. In *Asia and South Pacific Design Automation Conference*. 627–630.

ZHUANG, X. AND PANDE, S. 2003. Resolving Register Bank Conflicts for a Network Processor. In *International Conference on Parallel Architectures and Compilation Techniques*. 269–278.