A Compile-Time Managed Multi-Level Register File Hierarchy

Mark Gebhart[1]
mgebhart@cs.utexas.edu

Stephen W. Keckler[1,2]
skeckler@nvidia.com

William J. Dally[2,3]
bdally@nvidia.com

[1]The University of Texas at Austin
Austin, TX

[2]NVIDIA
Santa Clara, CA

[3]Stanford University
Stanford, CA

ABSTRACT

As processors increasingly become power limited, performance improvements will be achieved by rearchitecting systems with energy efficiency as the primary design constraint. While some of these optimizations will be hardware based, combined hardware and software techniques likely will be the most productive. This work redesigns the register file system of a modern throughput processor with a combined hardware and software solution that reduces register file energy without harming system performance. Throughput processors utilize a large number of threads to tolerate latency, requiring a large, energy-intensive register file to store thread context. Our results show that a compiler controlled register file hierarchy can reduce register file energy by up to 54%, compared to a hardware only caching approach that reduces register file energy by 34%. We explore register allocation algorithms that are specifically targeted to improve energy efficiency by sharing temporary register file resources across concurrently running threads and conduct a detailed limit study on the further potential to optimize operand delivery for throughput processors. Our efficiency gains represent a direct performance gain for power limited systems, such as GPUs.

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

General Terms: Experimentation, Measurement

1. INTRODUCTION

Modern GPUs contain tens of thousands of threads to tolerate main memory and function unit latencies. Each thread must store its register context in the processor's register file, leading to large register files that are energy intensive to access. For example, the latest NVIDIA GPU contains a 128KB register file per processor, with 16 processors per chip. The register file of such systems consumes 15–20% of the processor's dynamic energy, making it an attractive target for optimization [13].

GPU workloads present common register usage patterns. Most values are only read a single time, often within a few instructions of being produced. Values that are read several times tend to be read in bursts. Our prior work exploits these patterns with a register file cache (RFC) that reduces the number of accesses to the register file [11]. This technique was able to reduce register file system energy by 36%. In this work, we propose an alternate system in which the compiler explicitly controls operand movement through the register file hierarchy, using a operand register file (ORF) rather than an RFC. We also propose a deeper register file hierarchy, with an additional 1 entry per thread, last result file (LRF), which consumes minimal energy to access.

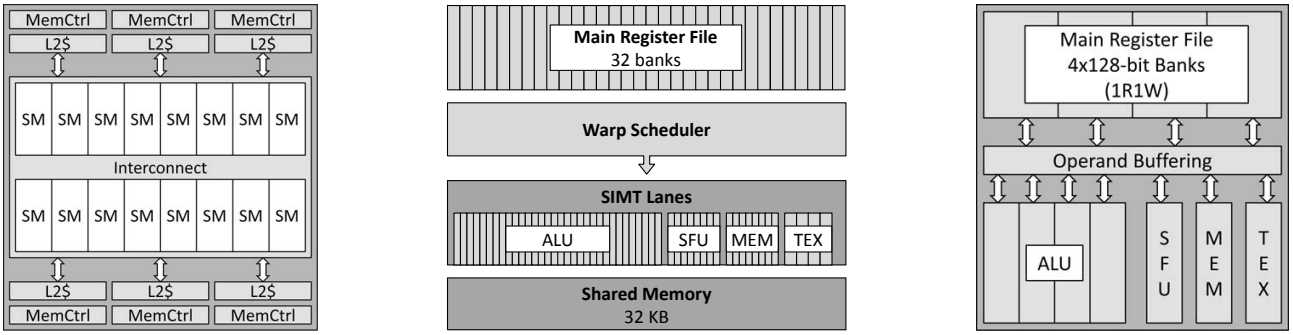
Allocating values across the register file hierarchy to minimize energy requires different algorithms than traditional register allocation. In traditional register allocation, allocation decisions focus on performance, due to the different access times of registers and memory [4, 21]. Since our baseline system is heavily pipelined to tolerate multi-cycle register file accesses, accessing operands from different levels of the register file hierarchy does not impact performance. Instead, each level of the hierarchy requires increasing amounts of energy to access. To reduce the storage requirements of the upper levels of the register file hierarchy, we use a two-level thread scheduler. The scheduler partitions threads into active and pending threads and only active threads are allowed to issue instructions and allocate entries in the ORF and LRF. Along with performing register allocation, the compiler moves threads between these two states.

Performing compiler allocation reduces reads to the main register file by 25% compared to the previous proposed RFC. Additionally, software control minimizes writes to the register file hierarchy. Deepening the register file hierarchy further reduces energy by replacing half of the accesses to the ORF with accesses to the LRF. Overall, our techniques reduce register file energy by 54%, an improvement of 44% compared to the hardware only RFC.

The remainder of this paper is organized as follows. Section 2 discusses background and prior work using an RFC. Section 3 describes our microarchitecture. Section 4 discusses the allocation algorithms. Section 5 and 6 discuss our methodology and results. Section 7 presents a limit study on opportunities to further optimize operand delivery. Sections 8 and 9 discuss related work and conclusions.

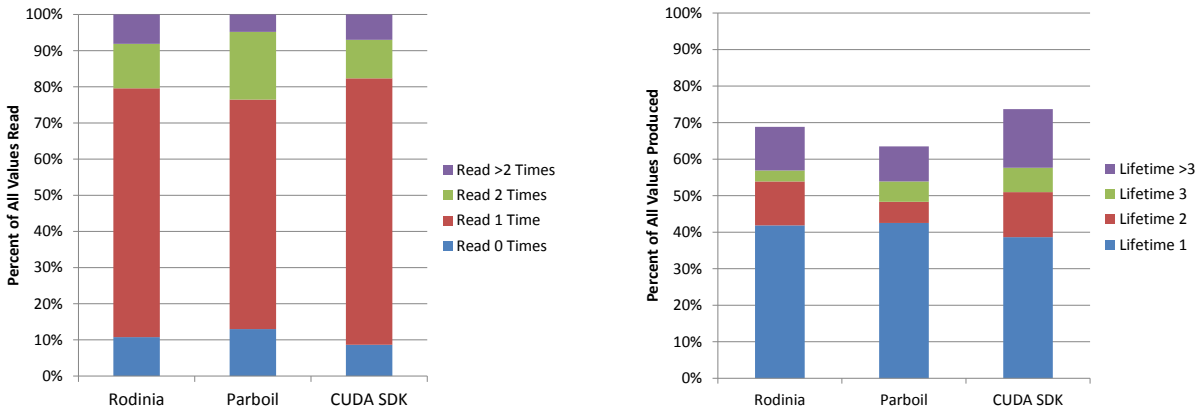
This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

MICRO'11, December 3–7, 2011, Porto Alegre, Brazil.



(a) GPU chip. (b) Streaming multiprocessor (SM). (c) 4-Wide SIMT cluster, replicated 8 times to form an SM.

Figure 1: Baseline GPU architecture.



(a) Register value usage. (b) Lifetime (instructions) of values only read once.

Figure 2: Register Usage Patterns

2. BACKGROUND

GPUs employ massive parallelism and multithreading to tolerate a range of latencies, including long latencies to DRAM. To increase efficiency, GPUs aggregate multiple threads into execution groups called warps. The threads within a warp are allowed to take their own execution paths, but performance is maximized when all threads in a warp take the same execution path. In this single instruction, multiple thread (SIMT) execution model, a single warp instruction is fetched each cycle and an active mask tracks which threads within a warp execute the fetched instruction. Our system, modeled after a contemporary GPU, contains a streaming multiprocessor (SM) that serves 32 warps of 32 threads each, for a total of 1024 machine resident threads per SM. Figures 1(a) and 1(b)show a chip-level and SM-level diagram of our baseline system. Each SM has access to 32KB of low latency, software-controlled, on-chip shared memory. Many applications first copy their working sets from global DRAM memory to the on-chip shared memory before operating on it, reducing the required number of DRAM accesses.

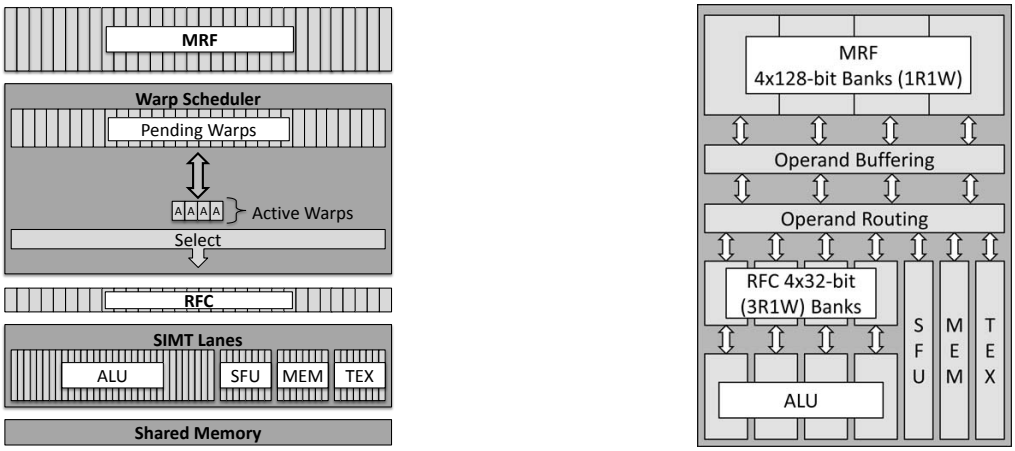
The main register file (MRF) stores context for all machine resident threads. The MRF is 128KB, allowing for 32 register entries per thread. The MRF is banked into 32, 4KB banks to provide high bandwidth. The operand buffering and distribution logic, shown in Figure 1(c), is responsible for fetching operands over several cycles and delivering them to the correct function unit. Each entry is 128 bits wide and stores a given register for 4 threads. The 128 bits must be split into 4, 32-bit values and distributed to the 4 SIMT lanes in a cluster, consuming a significant amount of wire energy. Each of the four lanes in the cluster has its own private ALU, but lanes share more expensive execution units including the special function unit (SFU for transcendental and other functions), the port to memory (MEM), and the texture unit (TEX - commonly used in graphics). The 4-wide SIMT lane cluster, shown in Figure 1(c) is replicated 8 times to form a 32-wide SM.

2.1 GPU Register Usage Patterns

Similar to CPU workloads [10], GPU workloads have common register usage patterns. Our prior work characterized these patterns for a set of NVIDIA-proprietary traces [11]. In this work, we replicate the experiment on a set of openly available applications. Figure 2(a) shows the number of times each value, written into the register file is read; Figure 2(b) shows the lifetime, in instructions, of values that are only read once. Up to 70% of values are only read once and 50% of all values produced are only read once, within three instructions of being produced. These characteristics can be leveraged to optimize register file system energy by keeping short lived values in small, low-energy structures, close to the function units that produce and consume them.

2.2 Prior Work: Register File Cache

In our prior work, we proposed a register file cache and a two-level warp scheduler to reduce register file energy [11]. Figure 3 shows that proposed architecture. In that work, we added a small, 6-entry per thread, register file cache (RFC)



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

Figure 3: Modified GPU microarchitecture from prior work [11]. (a) High level SM architecture: MRF with 32 128-bit banks, multiported RFC (3R/1W per lane). (b) Detailed SM microarchitecture: 4-lane cluster replicated 8 times to form 32 wide machine.

to each SIMT lane. Each bank of the RFC is private to a single SIMT lane. Values produced by the function units are written into the RFC. Instructions first check the RFC for their read operands and then only access the MRF for values not present in the cache. The RFC employs a FIFO replacement policy and writes evicted values to the MRF. To prevent writing dead values back to the MRF, the compiler encodes static liveness information in the program binary to elide writebacks of dead values to the MRF.

We also introduced a two-level warp scheduler to reduce the size of the RFC. The high thread count on a GPU hides two sources of latency. A large number of warps are needed to tolerate the long latencies from main memory access. However, the short latencies from function units and shared memory access can be tolerated with a much smaller set of threads. The two-level scheduler partitions threads into active threads that can issue instructions and pending threads that are waiting on long-latency operations. Only the active threads have RFC entries allocated. When an active warp encounters a dependence on a long latency event, it is swapped out and its RFC values are flushed to the MRF. The descheduled warp is replaced with a ready warp from the pending set. With at least 8 active warps, out of a total of 32 machine resident warps, the SM suffers no performance penalty from using the two-level warp scheduler. The RFC was able to reduce the number of MRF accesses by 40–70%, saving 36% of register file energy without a performance penalty. In this paper, we extend our prior work in two ways: (1) using software allocation of register resources instead of hardware-controlled caching, and (2) and deepening the register file hierarchy to further save energy.

3. MICROARCHITECTURE

This section describes our proposed microarchitecture, including moving from a hardware managed register file cache to a software managed operand register file and expanding the register file hierarchy to three levels.

3.1 ORF: Operand Register File

A hardware controlled RFC that captures the written register values of all executed instructions has two main inefficiencies. First, values that are evicted from the RFC consume energy being read from the RFC before being written to the MRF. Second the RFC must track the register names from the larger MRF namespace with tags and lookups in the RFC.

We propose a software managed operand register file (ORF) that eliminates these deficiencies. The compiler can leverage its knowledge of register usage patterns to determine where to allocate values across the register file hierarchy, mapping frequently read or soon to be read values to the ORF and values with less temporal locality directly to the MRF. Values that have temporal locality and that are persistent can be written to both the ORF and MRF in the same instruction, eliminating write-back on eviction.

Rather than add explicit bits to indicate the level of the register file hierarchy, the register file namespace is partitioned with each segment of architectural register names representing a different level of the hierarchy. As a result, this approach does not increase the energy associated with storing larger instructions or increase the instruction decode energy. Further, because the location of each operand is determined at decode time, the ORF need not employ tags or incur tag checking energy overheads. Because the architecture register namespace is typically under-utilized, using a portion of the namespace to represent LRF and ORF entries does not diminish a program's ability to fully use its MRF resources. Since the existing register file namespace is used, a binary can correctly execute on chips with different hierarchy organizations. However, as modern GPUs typically employ just-in-time compilation to perform chip-specific code-generation, the JIT compiler can perform optimizations for design specific register hierarchies.

3.2 LRF: Last Result File

A three-level register file hierarchy is more energy-efficient than a two-level hierarchy. Our results in Section 6.4 show that a hardware caching scheme benefits from a three-level register file hierarchy, but a software controlled hierarchy sees greater benefits, because the compiler can control data movement in the register file hierarchy to minimizing energy. Figure 4 shows our proposed three-level hierarchy for a 4-wide SIMT cluster. Our hierarchy consists of a small upper-level, last result file (LRF), a medium sized ORF, and a

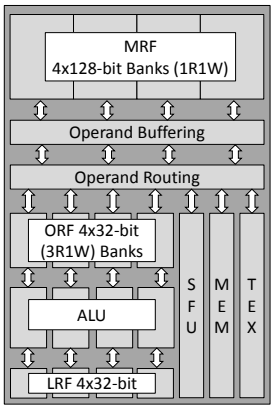


Figure 4: Microarchitecture of three-level register file organization for 4 SIMT lanes, the LRF is only accessible from the ALU units.

large lower-level MRF. Our value usage analysis found many values whose only consumer was the next instruction. By introducing an LRF, with only a single entry per thread, we capture these results in a very small structure with low access energy.

Each of the ORF and LRF entries is 32-bits wide and each bank has 3 read ports and 1 write port. Three read ports enables single cycle operand reads, eliminating the costly operand distribution and buffering required for MRF accesses. Our workloads use parallel thread execution (PTX) assembly code, which supports 64-bit and 128-bit values [16]. Values wider than 32-bits are stored across multiple 32-bit registers. In the workloads we examined, the vast majority of instructions (99.5%) only operate on 32-bit values. When a larger width value is encountered, the compiler allocates multiple entries to store the value in the ORF. The compiler encodes the level of the register file for each register operand in the instruction. Because the percent of instructions operating on wider data values in our workloads is small, this approach incurs negligible overhead from instructions with wide register operands.

The private ALUs operate with full warp wide throughput. The SFU, MEM, and TEX units operate at a reduced throughput and we collectively refer to them as the shared datapath. Our register profiling shows that only 7% of all values produced are consumed by the shared datapath. Further, 70% of the values consumed by the shared datapath are produced by the private datapath, reducing the opportunity to store values produced and consumed by the shared datapath near the shared datapath. Due to these access patterns, the most energy-efficient configuration is for the shared datapath to be able to access values from the ORF but not from the LRF. By restricting the LRF to only be accessible from the private datapath, we minimize the ALU to LRF wire path and the associated wire energy.

We also explore an alternative LRF design that splits the LRF 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 LRF design, rather than a single LRF bank per SIMT lane, each lane has a separate LRF bank for each of the three operand slots. Each of the three LRF banks per lane contains a single 32-bit entry. The compiler encodes which LRF bank a value should be written to and read from. This design increases the effective size of the LRF, while keeping the access energy minimal. Our results show that short-lived values stored in the LRF are not commonly consumed across different operand slots. When a value is consumed across separate operand slots, the compiler allocates it to the ORF, rather than to the split LRF. Using a split LRF design can increase wiring energy, a tradeoff we evaluate in Section 6.4.

4. ALLOCATION ALGORITHM

The compiler minimizes register file energy by both reducing the number of accesses to the MRF and by keeping values as close to the function 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. 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 LRF or ORF. Second, because the LRF and ORF are 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, which invalidates the LRF and 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 LRF and to a lesser extent the 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.

4.1 Extensions to Two-Level Warp Scheduler

The two-level warp scheduler used in our prior work with a 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 our 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 somewhat from usage in prior work by Crago et al., which splits a thread into separate memory accessing and memory consuming instruction streams termed strands [7]. Similar to this prior work, we use the concept of a strand to indicate a stream of instructions that terminates on a long-latency event.

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 6.5. 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 (BBs) 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

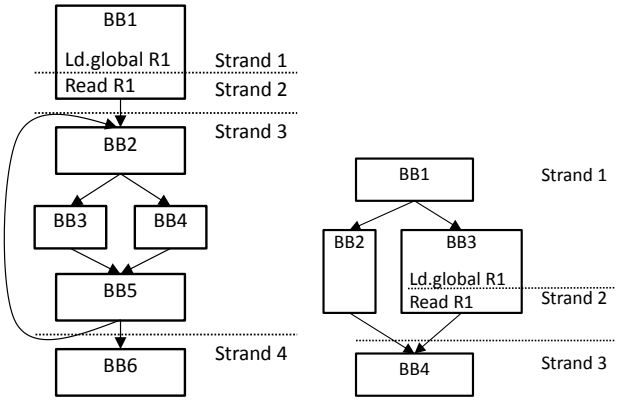


Figure 5: Examples of strand endpoints.

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

Figure 6: Function to calculate energy savings of allocating a register instance to the ORF.

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.

Figure 5(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 caused 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 LRF or ORF past strand boundaries. Section 7 explores relaxing the requirement that strands may not contain backward branches. Figure 5(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. This situation is rare such that these extra strand endpoints have negligible effect.

4.2 Baseline Algorithm

We first discuss a simplified version of our allocation algorithm that assumes a two-level register file hierarchy (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 [16]. 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 6. We calculate the

```
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
```

Figure 7: Algorithm for performing ORF allocation.

number of reads in the strand for each value and if each value is live-out of the strand. Values live-out of the strand 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 7 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 static instruction issue slots they would occupy the ORF. Scaling the energy savings by an approximation of the length of time the ORF 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 5.2. 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.

4.3 Partial Range Allocation

Figure 8(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

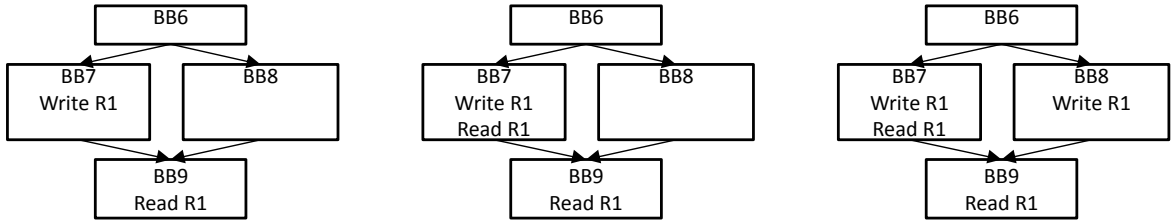


Figure 10: Examples of control flow effects on ORF allocation.

```
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

Figure 8: Examples of allocation optimizations.

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

Figure 9: Function to calculate energy savings of allocating a read operand to the ORF.

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 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 algorithm could perform sub-optimally if we allocate partial ranges after full ranges, as the ORF may already have been allocated to a value that saves less energy. However, our greedy algorithm performs well with partial range allocation converting a significant number of MRF reads into ORF reads.

4.4 Read Operand Allocation

Figure 8(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 9 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 8(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 7. 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.

4.5 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 values to remain in the ORF across forward branches. Figure 10 illustrates three different patterns that occur with forward branches, assuming that `R1` has been written to the MRF by a previous strand. In Figure 10(a), `R1` is written in `BB7` and consumed in `BB9`. However, because `BB8` does not write `R1` and `BB9` must statically encode from where `R1` should be read, `R1` cannot be allocated to the ORF. In Figure 10(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 10(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 10(c) are to the ORF, eliminating all MRF accesses. The compiler handles these three cases by ensuring that when there is uncertainty in a value's location due to control flow the value will always be available from the MRF.

4.6 Extending to Three-Level Hierarchy

We expand our baseline algorithm shown in Figure 7 to consider splitting values between the LRF, ORF, and MRF. When performing allocation, we first try to allocate as many values to the LRF as possible. Again, we prioritize values based on the energy savings of allocating them to the LRF divided by the number of instruction slots they occupy the LRF. Almost all values allocated to the LRF 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 ORF. Values that cannot be allocated to the ORF are shortened using our previously discussed partial range allocation algorithm; we then attempt to allocate these short-

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

Table 1: Benchmarks.

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 DRAM 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

Table 2: Simulation parameters.

Entries	Read Energy (pJ)	Write Energy (pJ)
1	0.7	2.0
2	1.2	3.8
3	1.2	4.4
4	1.9	6.1
5	2.0	6.0
6	2.0	6.7
7	2.4	7.7
8	3.4	10.9

Table 3: ORF energy.

Parameter	Value
MRF Read/Write Energy	8 / 11 pJ
LRF Read/Write Energy	0.7 / 2 pJ
MRF Bank Area	38000 μ^2
MRF Distance to Private	1 mm
ORF Distance to Private	0.2 mm
LRF Distance to Private	0.05 mm
MRF Distance to Shared	1 mm
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

Table 4: Modeling parameters.

ened ranges to the ORF. We explored allocating values to both the LRF and 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 LRF or the ORF but not both, simplifying the design. To minimize wire distance on the commonly traversed ALU to LRF path, we restrict the LRF to be accessed by only the private datapath. Therefore, the compiler must ensure that values accessed by the shared datapath (SFU, MEM, and TEX units) are only allocated into the ORF. As discussed in Section 3.2, we explore using a split LRF design, where each operand slot has a private LRF bank. With this design, the compiler encodes which LRF bank values should be written to and read from. Values that are accessed by more than one operand slot must be allocated to the ORF.

5. METHODOLOGY

5.1 Simulation

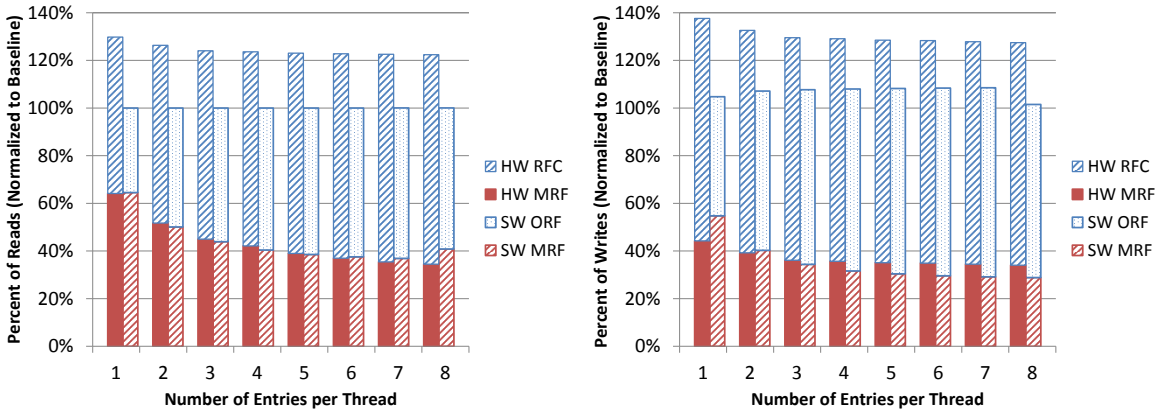
We use Ocelot, an open source, dynamic compilation framework for PTX to run our static register allocation pass on each kernel [9]. Ocelot provides useful compiler information such as dataflow analysis, control flow analysis, and dominance analysis. We augment Ocelot's internal representation of PTX to annotate each register access with the level of the hierarchy that the value should be read from or written to. We use benchmarks from the CUDA SDK [15], Rodinia [5], and Parboil [17], shown in Table 1. These benchmarks are indicative of compute applications designed for 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. The Parboil suite is design to exploit the massive parallelism available on GPUs and these applications generally have the longest execution times of all of our benchmarks.

We evaluate the energy savings of our register file hierarchies and the performance effects of the two-level warp scheduler for our benchmarks. To evaluate the energy savings we run each benchmark to completion, using the default arguments and input sets provided with Ocelot. We use a custom Ocelot trace analysis tool to record the number of accesses that occur to each level of the register file over the entire course of the program's execution. We exclude two benchmarks that take longer than 5 days to execute.

During the full application execution, we create a trace that specifies the execution frequency of each dynamic control flow path in the application. We built a custom trace based simulator that uses these traces to verify the performance effects of two-level scheduling, using the simulation parameters given in Table 2. Our simulator uses the execution frequencies to reconstruct likely warp interleavings that are encountered over the course of the full application. We execute 100 million thread instructions per benchmark on our trace based simulator to verify the performance effects of two-level scheduling. Since this work focuses on the register file architecture of an SM, we limit the simulation to a single SM rather than simulate an entire GPU.

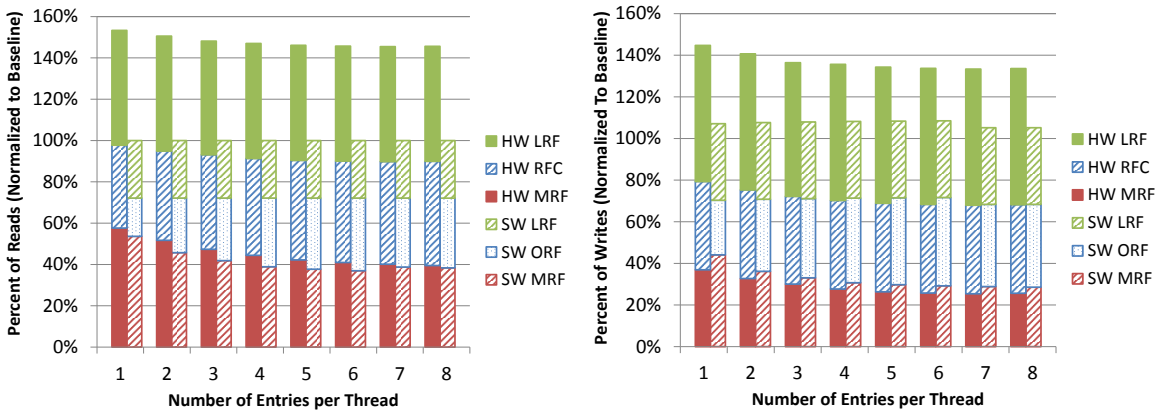
5.2 Energy Model

We model the ORF and LRF as 3-read port, 1-write port flip-flop arrays. We synthesize the designs using Synopsys Design Compiler with clock-gating and power optimizations enabled with a commercial 40 nm high-performance standard cell library. Our synthesis targets 1 GHz at 0.9V. Table 3 shows the energy required to access 128-bits from different sized ORFs, sized to support 8 active warps. We model the MRF as 128-bit wide SRAM banks with 1 read and 1 write port. We generate the SRAM banks using a commercial memory compiler and characterize their energy requirements at 1GHz. We use the methodology of [14] to model wire energy with the parameters shown in Table 4. Each ORF bank is private to a SIMT lane, greatly reducing the wiring energy. The LRF is only accessed by the ALUs in the private datapath, further reducing wiring energy. Compared to the MRF, the wire energy for the private datapath is reduced by a factor of 5 for accesses to the ORF and a factor of 20 for accesses to the LRF.



(a) Read Breakdown (b) Write Breakdown

Figure 11: Reads and writes to two-level register file hierarchy, normalized to single-level register file.



(a) Read Breakdown (b) Write Breakdown

Figure 12: Reads and writes to three-level register file hierarchy, normalized to single-level register file.

6. RESULTS

We present our initial results in a technology independent metric by showing the breakdown of reads and writes across the register file hierarchy. In Section 6.4, we combine our access count measurements and energy estimates to calculate the energy savings of our various schemes. First, we confirm prior work on two-level warp scheduling by finding no performance penalty when using a two-level warp scheduler with 8 active warps. All of our remaining results assume a two-level warp scheduler using 8 active warps, which only allocates LRF and ORF entries for active warps.

6.1 SW versus HW Control

Next, we compare our prior work using a HW controlled RFC with our most basic SW controlled ORF. Figure 11 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 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. Partial range allocation and read operand allocation reduce the number of MRF reads by 20% at the cost of increasing ORF writes by 8%. Section 6.4 shows that this tradeoff saves energy.

6.2 Deepening the Register File Hierarchy

Adding a small (1 entry / thread) LRF to the register file 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 LRF. When a value is evicted from the LRF, it is written back to the RFC; likewise when a value is evicted from the RFC, it is written back to the MRF. When a thread is descheduled, values in the LRF and RFC are written back to the MRF. We use static liveness information to inhibit the writeback of dead values. Because the shared function units cannot access the HW LRF, the compiler ensures that values accessed by the shared units will

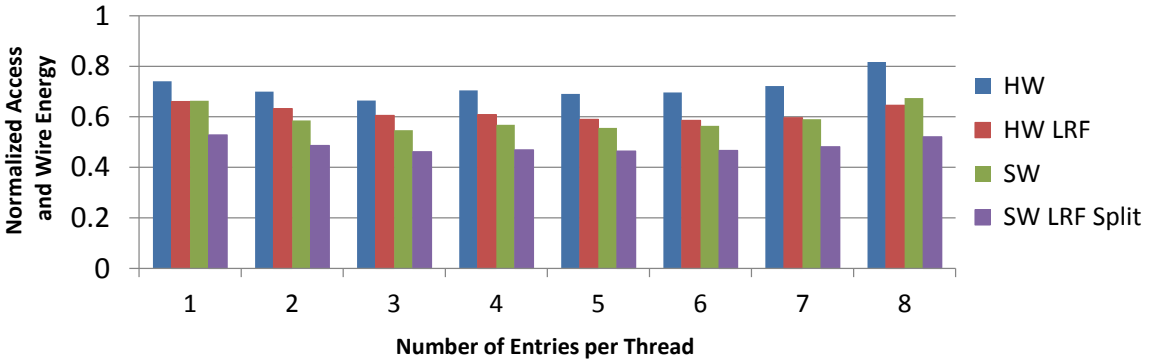


Figure 13: Energy savings of register file organizations, normalized to baseline with single level register file.

be available in the RFC or MRF. When using a software managed hierarchy, the compiler controls all data movement across the three levels.

Figure 12 shows the breakdown of reads and writes across the three levels of the 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 LRF and ORF. Despite its small size, the LRF still captures 30% of all reads, resulting in 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. Because of control flow uncertainties, the number of MRF writes increases slightly when using a software managed design, which presents a minimal energy overhead.

6.3 Split versus Unified LRF

Finally, we consider the effect of a design with a separate LRF bank for each operand slot. For example, a floating-point multiply-add (FMA) operation of D = A * B + C, reads operands from slot A, B, and C. Having a separate bank for each LRF operand slot allows an instruction, in the best case, to read all of its operands from the LRF. With a unified LRF, only a single operand can be read from the LRF and the others must be read from the ORF. A split LRF increases reads to the LRF by nearly 20%, leading to decreases in both ORF and MRF reads. Using a split LRF design has the potential to increase the wiring distance from the ALUs to the LRF, a tradeoff we evaluate in Section 6.4.

6.4 Energy Evaluation

We combine the access counts from the various configurations with our energy model to calculate the energy savings of the different options. Figure 13 shows the register bank access and wire energy normalized to a single level register file hierarchy. In our previous paper that proposed the RFC the register file energy savings were 36%. Our results for the HW controlled RFC, shown by the HW bar, show a 34% savings in register file energy, validating the prior result with a slight variance due to the different benchmarks. By relying on software allocation and optimizations to our baseline algorithm, we are able to improve this savings to 45%, as shown by the 3-entry per thread SW bar. The optimizations of partial range allocation and read operand allocation provide a 3–4% improvement in energy efficiency over the baseline allocation algorithm. The software bars in Figure 13 include these optimizations. Compared to the HW

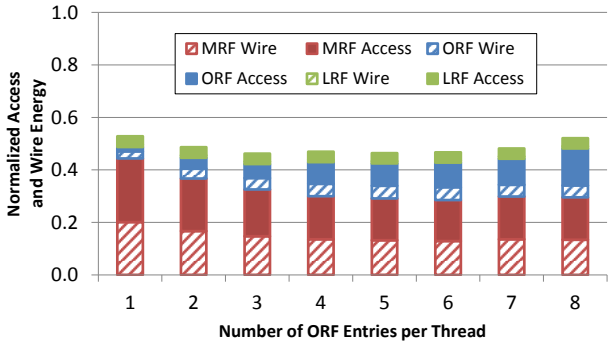


Figure 14: Energy breakdown of most efficient design, normalized to baseline single level register file.

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 LRF reduces energy for both the hardware and software controlled schemes, although the benefit is larger when using software control, shown by the HW LRF and SW LRF Split bars in Figure 13. The most energy-efficient three-level SW design uses 3 ORF entries per thread, saving 54% of register file energy, while the most energy-efficient three-level HW design uses 6 RFC entries per thread, saving 41% of register file energy. The compiler is better able to utilize each entry and the smaller structure reduces the per-access energy. Splitting the LRF provides a 4% energy reduction compared with a unified LRF.

Figure 14 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 ORF. Even though the LRF captures 1/3 of operand reads, accesses to the LRF consume a very small portion of the overall energy, due to its small size, motivating future work to focus on reducing MRF accesses. While a split LRF has the potential to increase LRF wire distance and energy, Figure 14 shows that the LRF wire energy comprises less than 1% of the baseline register file energy.

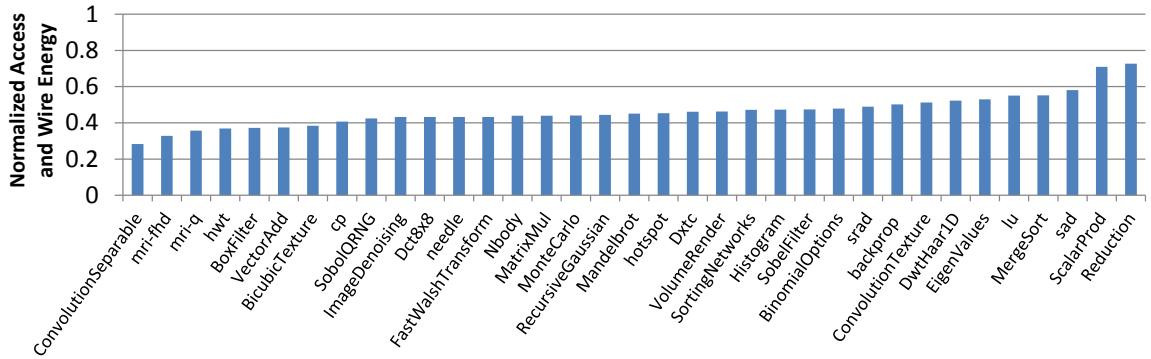


Figure 15: Per benchmark access and wire energy results for the most energy efficient configuration, sorted by energy savings. (3-entry ORF with split LRF using read operand allocation and partial range allocation.)

Figure 15 shows the per-benchmark normalized register file energy for our most energy-efficient design, the SW LRF Split configuration with 3 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 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 LRF and 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 LRF and ORF.

Our most energy-efficient three-level configuration saves 54% of register file energy, a 44% improvement over the prior HW register file cache and a 27% improvement over a purely HW controlled three-level design. Using our previously proposed high-level GPU power model [11], our reduction in register file energy represents a 8.3% reduction in SM dynamic power, which is a savings of 5.8% of chip-wide dynamic power. Our system suffers no performance penalty and simplifies the microarchitecture of our prior work by eliminating register file cache tags.

6.5 Instruction Encoding Overhead

Our proposed system makes two changes to instruction encodings: specifying the level of the hierarchy each operand is located and an additional bit to each instruction to indicate the end of a strand. While the cost of storing and encoding this information could potentially negate energy savings from the software controlled register file hierarchy, our analysis indicates that the overheads are quite low.

Prior work has found that on a GPU instruction fetch, decode, and schedule represents roughly 15% of chip-wide dynamic power. This percentage reflects the natural instruction fetch efficiency of GPUs in which a single instruction fetch can feed all of the instructions in a warp (up to 32 in our model). If we conservatively assume that fetch, decode, 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 effect of extra bits we make the simplifying assumption that additional bits result in linear increases in fetch and decode energy. On current generation NVIDIA GPUs there is unused space in the register file namespace that we use to encode hierarchy information. Therefore our overhead is only a single bit per instruction to indicate the end of a strand, resulting in a 3% increase in fetch and decode energy, a chip-wide overhead of 0.3%, and an overall increase of energy efficiency of 5.5%.

If we assume pessimistically that extra bits are required for each operand, an instruction may require as many as 5 additional bits: 4 to increase the register namespace for each of 4 operands and 1 to indicate the end of a strand. This assumption yields an increase in chip-wide fetch and decode energy of 15%, resulting in a 1.5% chip-wide energy overhead. This high-level model makes several simplifying assumptions and is intended to show that our proposed system would still save energy even with worst-case encoding overheads. A real-world system must carefully tradeoff the increase in fetch and decode energy versus the utility of this additional compiler provided data when designing the ISA. Even with these worst case assumptions, our proposed system reduces chip-wide dynamic power by at least 4.3%.

7. REGISTER HIERARCHY LIMIT STUDY

In this section, we consider several possible extensions to our work and their potential effects on energy efficiency. The most energy-efficient of our designs, with a three-level hierarchy, reduces register file energy by 54%. An ideal system where every access is to the LRF would reduce register file energy by 87%. Of course this system is not practical as the LRF is too small to hold the working set of registers and the LRF is not preserved across strand boundaries. If each operand only accessed a 5-entry per thread ORF, register file energy would be reduced by 61%. In a realistic system, each level of the hierarchy must be accessed, with the MRF holding values needed across strand boundaries and the ORF and LRF holding temporary values. Our current system performs well and is competitive with an idealized system, in which every access is to the ORF.

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. Using the oracle scheduler, this variable allocation policy 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 to know which threads would be scheduled in the future. Finally, running with fewer active threads has the potential to harm performance.

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.

Instruction Scheduling: Reordering instructions presents two opportunities to reduce register file energy. The first is to reorder instructions within a 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 a 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 is to increase 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 instruction relative to long latency events by never flushing the LRF or ORF when a warp is descheduled. In this idealized experiment, all machine resident warps, not just active warps, have LRF and 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.

8. RELATED WORK

Prior work has explored improving register file's efficiency by reducing the number of entries [2, 27], reducing the number of ports [18], and reducing the number of accesses [20, 25]. These approaches have been explored for traditional CPUs, VLIW processors [28, 29, 30], and streaming processors [8, 22]. Several systems, as early as the CRAY-1, have proposed a compiler controlled register file hierarchy to improve performance, energy, or area [6, 23]. Swensen and Patt show that a two-level register file hierarchy can provide nearly all of the performance benefit of a large register file on scientific codes [24]. Recent work, proposes using a hybrid SRAM / embedded DRAM (eDRAM) register file to reduce the register file access energy of a GPU [27]. They propose changes to the thread scheduler to minimize the effect 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. Using a variable number of registers, depending on a thread's usage, was proposed by [26].

ELM uses a software controlled two-level register file to reduce operand energy [3]. 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 [19].

AMD GPUs allocate short lived values to clause temporary registers [1]. These registers are managed by software and are not persistent across forward branches, unlike our ORF and LRF. The result of the last instruction can be accessed by a specially named register, similar to our LRF. Unlike our LRF, writes to this register cannot be inhibited so values can only be passed to the next instruction. Ayala describes a combined hardware / software technique to detect portions of a program where a small number of register file entries are needed and the unneeded entries are disabled in hardware to save access energy [2]. Gebotys performs low-energy register and memory allocation by solving a minimum cost network flow [12]. This approach requires a complicated algorithm for allocation and we find little to no opportunity to improve the allocations resulting from our greedy algorithm. Section 2.2 provides an overview of our previous work using a hardware controlled register file cache and a two-level thread scheduler.

9. CONCLUSION

Energy efficiency must be optimized for all future computer systems, ranging from low-powered embedded devices to high-powered server-class processors. Many of the traditional structures on a chip present opportunities to be rearchitected to improve energy efficiency. In this work, we redesign the register file system of a modern GPU to utilize a three-level hierarchy where all data movement is orchestrated at compile-time. We explore compiler algorithms to share temporary register file resources across concurrently executing threads. This combined hardware and software approach reduces register file system power by 54%, without harming performance, and represents a 44% improvement over the savings achieved by a previously proposed hardware controlled RFC.

Acknowledgments

We thank the anonymous reviewers, Daniel Johnson, Jeff Diamond, 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.

10. REFERENCES

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

[2] J. L. Ayala, A. Veidenbaum, and M. Lopez-Vallejo. Power-aware Compilation for Register File Energy Reduction. *International Journal of Parallel Programming*, 31(6):451–467, December 2003.

[3] J. Balfour, R. Harting, and W. Dally. Operand Registers and Explicit Operand Forwarding. *IEEE Computer Architecture Letters*, 8(2):60–63, July 2009.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6:47–57, 1981.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization*, pages 44–54, October 2009.

[6] K. D. Cooper and T. J. Harvey. Compiler-controlled Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1998.

[7] N. Crago and S. Patel. OUTRIDER: Efficient Memory Latency Tolerance with Decoupled Strands. In *International Symposium on Computer Architecture*, pages 117–128, June 2011.

[8] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. In *International Conference for High Performance Computing*, pages 35–42, November 2003.

[9] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 353 – 364, September 2010.

[10] M. Franklin and G. S. Sohi. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-grain Parallel Processors. In *International Symposium on Microarchitecture*, pages 236–245, November 1992.

[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. In *International Symposium on Computer Architecture*, pages 235–246, June 2011.

[12] C. H. Gebotys. Low Energy Memory and Register Allocation Using Network Flow. In *Design Automation Conference*, pages 435–440, June 1997.

[13] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *International Symposium on Computer Architecture*, pages 280–289, June 2010.

[14] P. Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report TR-2008-13, University of Notre Dame, September 2008.

[15] NVIDIA. 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`, June 2008.

[16] NVIDIA. 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`, 2011.

[17] Parboil Benchmark Suite. `http://impact.crhc.illinois.edu/parboil.php`.

[18] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing Register Ports for Higher Speed and Lower Energy. In *International Symposium on Microarchitecture*, pages 171–182, December 2002.

[19] J. Park and W. J. Dally. Guaranteeing Forward Progress of Unified Register Allocation and Instruction Scheduling. Technical Report Concurrent VLSI Architecture Group Memo 127, Stanford University, March 2011.

[20] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass Aware Instruction Scheduling for Register File Power Reduction. In *Languages, Compilers and Tools for Embedded Systems*, pages 173–181, April 2006.

[21] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Language Systems*, 21(5):895–913, September 1999.

[22] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register Organization for Media Processing. In *International Symposium on High Performance Computer Architecture*, pages 375–386, 2000.

[23] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[24] J. A. Swensen and Y. N. Patt. Hierarchical Registers for Scientific Computers. In *International Conference on Supercomputing*, pages 346–354, September 1988.

[25] J. H. Tseng and K. Asanovic. Energy-Efficient Register Access. In *Symposium on Integrated Circuits and Systems Design*, pages 377–382, September 2000.

[26] M. N. Yankelevsky and C. D. Polychronopoulos. α-Coral: A Multigrain, Multithreaded Processor Architecture. In *International Conference on Supercomputing*, pages 358–367, June 2001.

[27] W. Yu, R. Huang, S. Xu, S.-E. Wang, E. Kan, and G. E. Suh. SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading. In *International Symposium on Computer Architecture*, June 2011.

[28] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Two-level Hierarchical Register File Organization for VLIW Processors. In *International Symposium on Microarchitecture*, pages 137–146, December 2000.

[29] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Software and Hardware Techniques to Optimize Register File Utilization in VLIW Architectures. *International Journal of Parallel Programming*, 32(6):447–474, December 2004.

[30] Y. Zhang, H. he, and Y. Sin. A New Register File Access Architecture for Software Pipelining in VLIW Processors. In *Asia and South Pacific Design Automation Conference*, pages 627–630, January 2005.