UNIVERSITY OF CALIFORNIA

SANTA CRUZ

DYNAMIC INSTRUCTION FUSION

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

By

Ian Lee

December 2012

The Thesis of Ian Lee is approved:

Prof. Jose Renau, Chair

Prof. Matthew Guthaus

Prof. Cormac Flanigan

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Dynamic Instruction Fusion

By

Ian Lee

Energy efficiency in modern microprocessor design is a first order concern. Every facet of the microprocessor needs to be optimized now to be efficient in accesses, storage, and instruction execution. Dynamic Instruction Fusion provides a means to accomplish all three of these goals. By leveraging register re-use within typical instruction streams, whether generated through the use of a trace cache, or through wide issue instruction logic, it is possible to simultaneously reduce both the number of accesses to the register file, as well the number of instructions stored within the instruction window.

On average, Dynamic Instruction Fusion can reduce the number of instructions scheduled by ~ 48%, while simultaneously reducing the number of accesses to the register file by ~30%. This reduction in both the number of register file accesses and instruction window entries directly corresponds to a saving in energy in the register file.

Dedication

I would like to first thank my parents: Dawn and David, for all of the love and support that they have provided over the years. It hasn’t always been an easy journey, but they have been behind me 100% of the way. The friendships that I have made through graduate school at UC Santa Cruz are lifelong. Since first moving out here to California, my friends have become my family and I have been lucky enough to be able to bounce many ideas and thoughts off of them which led me to complete this thesis. In particular I want to thank Benjamin LaCara for his support and help with all of the long nights and weekends, and especially the coffee shop days this thesis was born out of. My advisor, Professor Jose Renau, has been a great source of knowledge and guidance these past few years. His understanding of computer architecture is remarkable and he has always been available for those 3 am questions. Of course, reminding all of us in the MASC lab that “it’s easy.”

Chapter 1 – Introduction

Today's modern High Performance Computing (HPC) applications continue to operate on larger data sets. In many cases, the same sequences of instructions are executed repeatedly hundreds or thousands of times. Many of these instructions are formed into tight loops performing the same operations multiple times. Consider the vector multiplication kernel, SAXPY, which multiplies a vector, A[], by a scalar, S, and adds a constant, C, to it. The pseudo-code for this kernel is as follows:

Algorithm 1: SAXPY Vector Multiplication

```
Input:  A[N], S, C
Output: Z[N]

        for (i = 0; i < N; i++) {
                Z[i] = A[i] * S + C;
        }
```

The assembly instructions executed for this code would be relatively simple as the scalar and constant values do not change between iterations of the loop, with only the loop counter incrementing, which the load and store addresses for the vectors.

When these loops are compiled, assembly code is produced, generating instructions of the form: X + Y => Z. Here, X and Y correspond to the source registers and Z corresponds to the destination register where the result will be stored. For simplicity a simple addition is depicted here, however it could be any type of instruction from the Instruction Set Architecture (ISA) which take 0, 1, or 2 sources and produce 0 or 1 destination register. Each access to these registers, whether reading or writing their values, costs a certain amount of energy based on the design of the register file.

After the instruction is fetched and its operands are decoded, it is placed into the instruction window where it waits for an available Functional Unit (FU) as well as any pending dependent operands to become ready. Typically, the instruction window is fairly large. Modern Intel architectures have instruction windows on the order of 128 entries, maintaining instructions ready for execution whenever a functional unit becomes available. While providing larger instruction windows, on the order of 1k entries, can provide a performance improvement by helping to hide main-memory latency, there is a penalty to using them in the form of the energy consumed by such a large memory structure [1].

The size of any memory structure: RAM, register file, instruction window, etc. directly affects the amount of energy that is required both to sustain the values in the structure (leakage energy), as well as to access the elements of the structure (dynamic energy). The main purpose of this thesis is to present a dynamic, hardware based methodology to reduce not only the number of registers accessed by the instruction stream, but also the total number of instructions which are issued through the instruction window. This is accomplished by examining streams of in-flight instructions, which are allowed to span multiple basic blocks (branch boundaries), and fusing similar types of instructions together. This fusion clusters several instructions of the same type into a single entry in the instruction window, to be processed by a specialized ALU.

Results show that this technique is able to reduce the number of source register reads by 32% and destination register writes by 25%. At the same time, this technique provides a 48% reduction to the number of instructions issuing through the pipeline through the use of highly fused instructions.

Chapter 2 – Related Works

Instruction fusion is not a new concept; however, until now it has never been applied at a level greater than looking at a few instructions at a time. Because there are such strong dependencies between instructions it is natural to look for ways to optimize the instruction stream by combining instructions and in particular, dependent instructions. Typically instructions are “fused” by considering groups of instructions to be the basic building block rather than taking each instruction individually. These groups of instructions can be executed by more powerful execution engines, such as those capable of performing arithmetic on three sources. In order to determine how to optimize these instructions streams, there must be a mechanism to allow the examination of greater numbers of instructions at once. One option is to leverage wide-issue processors, while another is to examine instruction streams which we expect to execute again, off the critical path of the execution engine.

2.1 Trace Cache

In order to perform the instruction fusion, we ideally want more instructions than are fetched each cycle by the typical modern processor. One method to obtain these instruction streams is through use of a trace cache [2] [3] [4]. Traces caches have been developed in order to take advantage of tight and often repeated loops. As instructions issue through the processor pipeline, they are analyzed and collected into trace lines which are stored into the trace cache for future use. Later in time, when the processor requests the same instruction address as an entry in the trace cache, the trace line will be fetched rather than the stream of instructions from the instruction cache.

The typical trace cache design allows for a trace line with as many as 16 instructions across 3 branches (4 basic blocks maximum). Returns, indirect branching, and serializing instructions terminate the creation of a trace line. Once a trace line has been finalized, it is stored into the trace cache. Some researchers [4] [3] have proposed that this is an ideal time to perform several dynamic optimizations to the trace. These optimizations include: instruction combination on a limited number of instructions, dependency collapsing, and dead code elimination.

One of the drawbacks to these traces is that they rely on the fact that the branches within the trace are predicted properly. Within long code loops, this will tend to be the case, with modern branch predictors providing very high prediction rates. However, eventually the loop will complete, and the execution must continue at a different point. This transition period can occur inside of a trace line which was being executed. Therefore, even though the assumption is made that a trace “knows” the taken/not-taken behavior of all branches within it, the branches must still be executed in order to verify. In the event that a branch miss-prediction was made, the in-flight instructions fetched from the trace cache must be flushed, the state rolled back, and the stream of instructions must be re-fetched, this time directly from the instruction cache. The benefits of the trace cache therefore rely on the accuracy of the branch predictor, and whether the execution path stored in the trace cache is correct. Figure 1 shows the branch prediction rates observed average 89.4% across all benchmarks, utilizing the O-GEHL branch predictor [5]. If the outliers, hmmer and sjeng are ignored, we see a 94.8% prediction rate, implying that we will be able to make effective use of a trace cache or more complex architecture (described later).

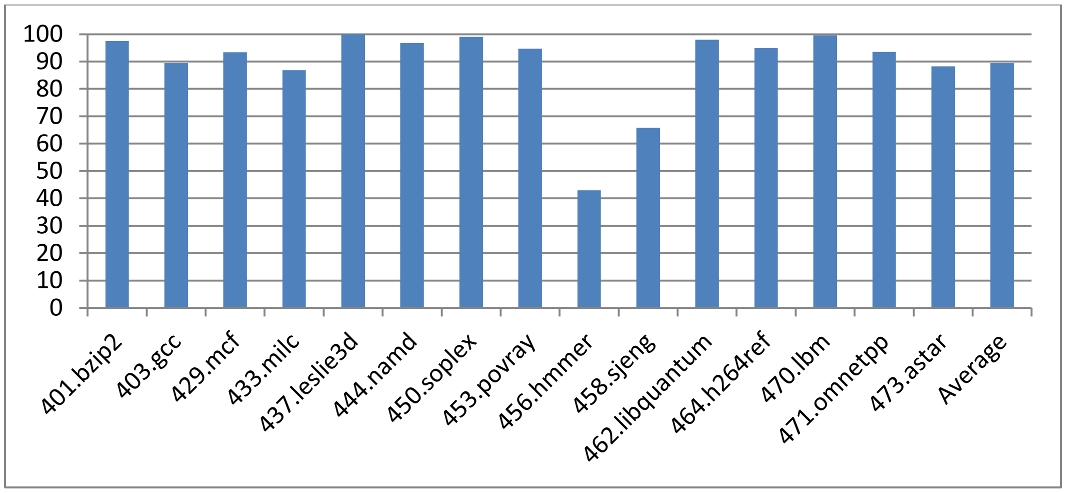

Figure 1: Branch Prediction Rates for OGEHL Branch Predictor

2.2 Instruction Fusion & Complex ALUs

Several researchers have proposed techniques to fuse instructions together in order to reach higher levels of performance [6] [7] [8] [9] [10]. Often times these techniques require the use of specialized ALUs which can handle the processing of these fused instructions.

Bracy et al. [6] proposed the use of what they called "Dataflow Mini-Graphs" to statically combine multiple instructions together. This approach provides a 2-12% performance improvement overall, with peak gains exceeding

40%. The processor deals with instructions through quasi-instructions called “handles.” These handles maintain the restriction that they be limited to two reads & one write, using a chain of ALUs to “amplify” the execution stage of the pipeline. In all stages of the pipeline, the handle is treated as a simple singleton instruction, except in the execution stage, where a dynamic instruction stream editor (DISE) consults a Mini-Graph Table to expand the instruction for execution. The ALU pipeline is a single entry, single exit chain of ALUs which selects its output from any single stage in the pipeline, which is important as it allows the substitution of ALU pipelines for ALUs without penalizing singleton instruction performance. The authors find that the majority of the improvements (60% coverage) are gained by using only 2 instruction mini-graphs, with some advantage being gained by using 3 or 4 instruction mini-graphs.

MACRO-OP Scheduling [7], proposed by Kim and Lipasti, is an alternative approach to pipelined ALUs which allows an out-of-order processor to dynamically combine pairs of dependent instructions. Within deeply pipelined processors there is a performance gap between memory and the processor core. In order to overcome this gap a large instruction window is used to hide the latency of memory operations. Increased instruction window size, however, requires complicated scheduling logic, while increasing the amount of energy consumed by the processor. MACRO-OP scheduling allows for pairs of dependent instructions to be combined, allowing for a reduction in the number of entries in the instruction window. For simplicity, the authors choose to limit the

MACRO-OP combinations to two dependent instructions, therefore having a maximum of 3 sources. Single cycle ALU operations, store address generation instructions, and control (branching) instructions are allowed to be combined in this way.

A similar idea is proposed by Vassiliadis et al. [8] through the use of what they call Interlock Collapsing ALUs. This specialized ALU is capable of performing 2's complement, unsigned binary, and binary logical operations, taking three sources and producing one output. Such an ALU allows instructions which would normally not be allowed to execute in the same cycle, creating "multi-operation instructions," such as:

$$R3 = R3 + R2$$

$$R4 = R4 + R3 \qquad => \qquad R4 = R4 + R3 + R2.$$

In modern simultaneous multithreading (SMT) processors, register availability is an important design consideration. Tran et al. [9] propose two simple techniques to provide register sharing. First, they allow dynamically combining physical registers which contain the same value. Second, they allow sharing of physical register storage among instructions modifying the same logical registers. In order to reduce the complexity of sharing registers with any arbitrary value, the authors limit themselves to sharing trivial values 0 and 1, as well as trivial computations which produce known 0 or X values (ie: X AND 0 => 0). They find that even with this limitation, they are able to produce 50% of the benefit as they would see if they allowed the sharing of arbitrary register values.

In the case of SMT processor where register availability is a bottleneck, the authors find similar performance between a regular processor with 200 physical registers and a modified processor using their sharing techniques but with only 160 physical registers.

Static Strands [10] are proposed by Sassone et al. as yet another means to dynamically collapse instructions in the pipeline. The authors observe that values generated within the strand tend to be “transient operands” that feed only a single dependent instruction. As many as 90% of the instruction strands the authors construct are purely ALU instructions with intermediate values which never leave the ALU. Therefore, these values have no need to be written into the register file, saving energy in the bypass logic (17-20% energy savings), issue logic (16-24% energy reduction), and register file accesses (13-14% energy reduction). In addition the authors are able to obtain a 15% increase in the number of Instructions Per Cycle (IPC), which they point out could be traded for additional increased energy savings by reducing the clock frequency while still maintaining the base performance. In contrast, the work presented in this thesis provides a general framework to allow handling instruction streams which generate more than a single destination value and which look beyond basic block boundaries, thereby providing greater energy savings.

Chapter 3 – Methodology

3.1 Fusion Selection Engine

Traditionally, instructions are fetched into the processor pipeline by way of the instruction cache, with the Program Counter (PC) specifying the current location in the executing code. Each cycle, instructions are fetched from the instruction cache, or alternatively, for those processors equipped with one, the PC can index into the trace cache to fetch a sequence of instructions that has been constructed into a trace. Modern processors are superscalar, meaning that they can fetch and decode multiple instructions per cycle. This behavior is quite like a trace cache, enabling Dynamic Instruction Fusion on these streams of instructions. Taking things a step further, it is possible to design a wide issue, superscalar processor which would be capable of predicting multiple branches in a single cycle. Such a feature should enable fetching the equivalent of a trace line worth of instructions: 16 instructions spanning three predicted branches, in order to match the specifications of a trace cache.

Once a stream of instructions becomes available, either from a trace cache or a wide issue processor capable of predicting multiple branches per cycle, it is passed to the Fusion Selection Engine (FSE). The FSE examines these instructions to determine how the fused instructions will be generated. There are two main approaches that you could leverage. Given the entire stream of as many as 16 instructions, we could try to compute the most optimal compression. This optimal case could be in terms of the number of read or write accesses to the register file, the total number of accesses to the register file, or the total number of instructions placed into the instruction window after the fusion. For the purposes of this work the energy consumption of a read and a write to the register file will be assumed to be the same, thus reducing the problem to a trade-off between the number of register file accesses (reads + writes) and the number of instructions required.

The difficulty with this approach is that solving the optimization on a given batch of instructions requires a trade-off between the number of register file accesses and the number of instructions which will go to the instruction window. In particular, the more varied the instruction mix, the more options there are for the fusion. An alternate approach is to simply examine the instructions in order, with a set of rules to determine how they should be fused. This is the preferred approach for this work, due to the goal of designing a low complexity hardware implementation for the proposed Dynamic Instruction Fusion.

The mix of instructions issuing through the pipeline plays a huge role in how efficiently the FSE can reduce the number of instructions and register file accesses. The instruction mix within the processor pipeline can be logically binned into one of five types of instructions: ALU, branch, load, store, or FPU. The count, as well as the mix of instructions which are considered for fusion, is given in Figure 2: Average Instruction Mix. ALU instructions (59.02%) are the primary type of instructions which are fused and for simplicity we combine branching instructions (7.74%) into ALU instructions, as they are both executed in the same way in the processor, as supported by Kim and Lipasti [7]. Load (15.55%) and store (15.76%) instructions operate exactly as you would expect, however, special care must be taken with them to avoid violating dependency rules. FPU instructions (1.93%) are similar to the ALU instructions, and could have their own fusion operations performed on them. For this work, the fusion of FPU instructions is not handled, with the focus instead on integer performance and applications, though it is extend this work to support them.

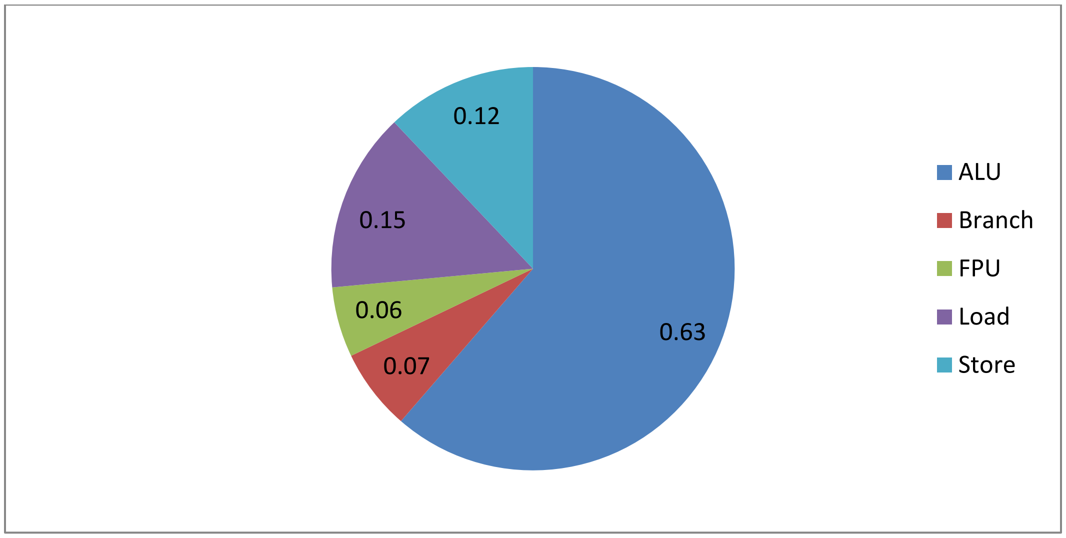

Figure 2: Average Instruction Mix

The most straightforward implementation of the FSE would be to insert the control logic between the fetch and the issue stages of the processor pipeline. This would allow for the processing of the instructions in an In-Order fashion, and therefore isolate the FSE from the more complicated Out-of-Order execution engine. An alternative placement would be applicable to trace cache enhanced processors, where the FSE could be combined into the trace construction logic.

For the purposes of this discussion, the assumption is made that the fusion occurs dynamically as instructions are fetched through the pipeline. In the alternate case that a trace cache is utilized, the fusion logic can be removed from the pipeline and run in the background as part of the trace formation process. Doing so has the benefit of offloading the processing from the critical path, therefore eliminating any delay that would be associated with the FSE, trading instead for a greater latency before the fusion instructions become available.

3.2 Standard Execution

In a standard processor, instructions are fetched from the instruction cache according to the current Program Counter (PC) value. These instructions are then issued to the execution units in the processor either in- or out-of-order. In-order execution refers to the instructions being issued in the same, sequential order which they were fetched. Out-of-order execution, on the other hand, provides the capability for instructions to be re-organized in such a way that instructions without pending dependencies may execute ahead of earlier fetched instructions. To guarantee correctness, the requirement that instructions retire in-order is enforced.

3.3 Naïve In-Order Fusion

The most basic approach to instruction fusion would be to operate the FSE on instructions, in-order, as they are fetched from the instruction cache. Naïvely, it makes sense to fuse all consecutive instructions together, as long as the stream is not interrupted by incoming load, store, or FPU operations. This particular approach is incredibly easy to implement, with an algorithm as follows:

Algorithm 2: Fusion Selection Engine - Naïve In-Order Handling

```
1. Initialize a new Fusion Instruction
2. FOR ALL instructions in Instruction Cache
    a. Fetch the instruction and decode the registers
    b. IF this instruction is an ALU operation
        i. Fuse the instruction
    c. ELSE
        i. Issue the Fusion Instruction
        ii. Issue the newly fetched, non-ALU, instruction
        iii. Initialize a new Fusion Instruction
```

In Dynamic Instruction Fusion, instructions selected to be fused by the FSE are not actually modified by the processor. The “fusion” itself comes from combining multiple instructions together and treating them as a single unit. This approach is very similar to the way that instructions are grouped together into a table and accessed by a “handle” in the Dataflow Mini-Graphs proposed by Bracy et al [6]. They are clustered together and passed through the execution engine as a unit, allowing for a reduction in the number of instructions which need to be managed individually. A detailed overview of the architecture required for this technique is described in Chapter 3.6 Required Architecture.

3.4 Queued Fusion

A simple, dynamic, in-order algorithm in the Fusion Selection Engine allows for greater efficiency in the fusion results, at the cost of increased complexity. This method does not build the best fusion stream possible, but provides a benefit in that it does not require a complicated decision engine for selecting the instructions to fuse, as would be required to squeeze the last bit of performance into the best fusions.

The queued fusion method operates by examining each instruction in order and making a decision based on rules for its instruction type. The assumption is made that fusing as many instructions as possible into as few fusion instructions as possible, is ideal. To this end, we apply the simplification that at any time there can be at most one single fusion instruction under construction. In the equations below, the syntax “Ra, Rb $\leftarrow$ Rx, Ry, Rz” is used to signify that the instruction takes in sources Rx, Ry, and Rz, operates on them, and then stores destination results in registers Ra and Rb. The operations that are performed are the same operations as the original un-fused instructions, modifying only the way in which the operations and registers are accessed. The rules for processing each instruction type are as follows:

If the current ALU instruction being examined requires a value from a pending load instruction, then we must issue the load instruction before proceeding with the current instruction. If it turns out that said load instruction had a dependence on a register within the currently pending fusion instruction, then we must also issue that fusion instruction. Example 1 is an example of this where the load (instruction 3) will cause the pending fusion operation to issue. In a similar fashion, if there is a pending store instruction and the current ALU instruction being evaluated would overwrite the register containing the value which we intend to store, we must issue both the store instruction as well as the fusion instruction which the store depends upon.

Load and store instructions obviously act in a similar fashion to the ALU instructions. If the current instruction being considered for fusion is either a load or store, we must check to ensure that any dependencies with instructions within the pending fusion instruction are maintained. In the case of a load, the new instruction may be attempting to overwrite a value needed by a pending fusion instruction. In Example 1, the load into R4 (instruction 3) would overwrite the value needed by the pending fusion instruction (instruction 2); therefore the naïve in-order approach to handle this case might be to simply mark the pending instruction as completed and send both it and the load, in order, to the execution pipeline. Another, more intelligent approach, would be to queue the load until we detect either that the destination of the load (R4) is needed, or that we are going to overwrite the source of the load (R1). These two cases can be seen as instructions 5 and 6 below, respectively.

Example 1: Fusion Selection Engine – Simple Load/Store/FPU Skipping

1. R2 ← R2, R4	*new fusion*	
2. R5 ← R2, R7	*fuses to*	*R2, R5 ← R2, R4, R7*
3. R4 ← LD[R1]		*R4 ← LD[R1]*
4. R2 ← R2, R7	*new fusion*	
5. R2 ← R2, R4	*fuses to*	*R2 ← R2, R4, R7*
6. R1 ← R2, R3	*fuses to*	*R1, R2 ← R2, R3, R4, R7*

In the case of the former, depending on the particular dependencies for the pending fusion instruction it may be possible to issue the load out of order, before the ALU instruction. In the case of this particular instruction stream, we are able delay the fusion by one instruction, providing a savings of an additional register read.

Example 2: FSE Load – Load Queue Handling

1. R2 ← R2, R4	*new fusion*	
2. R5 ← R2, R7	*fuses to*	*R2, R5 ← R2, R4, R7*
3. R4 ← LD[R1]	*queued*	*R4 ← LD[R1]*
4. R2 ← R2, R7	*fuses to*	*R2, R5 ← R2, R4, R7*
5. R2 ← R2, R4	*new fusion, issue queued load*	
6. R1 ← R2, R3	*fuses to*	*R1, R2 ← R2, R3, R4*

Example 1 provides an example of the naïve in-order load handling, while Example 2 shows how the load queue can be used to improve the fusion performance. The instructions displayed in red are the actual fused instructions issued from the FSE. In the original instruction stream there are 11 register reads, 6 writes, and 6 issued instructions (Example 1). The naïve, in-order fusion approach results in 8 reads, 5 writes, and 3 issued instructions (Example 2). By introducing the load queue, we are able to eliminate another of the register reads leaving: 7 reads, 5 writes, and 3 instructions issued as the optimal fusion.

Store instructions require a similar, though simpler, handling. With store instructions, there is no result produced, so we are solely concerned with whether the source register required by the store is going to be overwritten by an incoming ALU or Load instruction. As in the case of load instructions, store instructions can be handled either naïvely or with the added benefit of a store queue. The results for including both the load queue and the store queue are discussed further in Chapter 4 – Results.

For the purposes of this work, the focus is on designing around, and optimizing for, ALU operations. Floating point instructions are therefore handled similarly to the naïve approaches for load and store instructions. That is, they are simply passed through the FSE and issued to the execution engine. It would be possible to add a simple queue which would operate the same as the Load/Store Queues; however, due to the limited number of FPU instructions in the benchmarks examined, the reduction opportunities are limited.

3.5 "Perfect" Fusion

Using a mechanism such as a trace cache to feed the FSE provides certain benefits, including the ability to obtain a theoretically "perfect" fusion. This is due to the fact that by fetching a trace line from the cache, we have access to all information about all the instructions contained, which we wish to fuse. The difficulty comes from the fact that there are theoretically as many as 16! = ~$2x10^{13}$ possible instruction combinations to consider. In reality the number of viable combinations will be greatly reduced by the fact that we must continue to enforce dependencies between instructions. For comparison purposes, it would be useful to compare the results of the fusion algorithms here to some theoretical baseline. Due to the difficulty in determining the definition of "perfect fusion," two separate perfect baselines are created.

The first corresponds to if there was perfect register file access. In this baseline the key metric is the number of unique registers accessed. This is sub-divided to consider the number of unique sources as well as unique destinations. Given an infinitely powerful computational unit, this would correspond to the minimum number of register file accesses possible.

The second baseline corresponds to the minimum number of instructions that could be issued. The fusion techniques proposed in this thesis provide support only for ALU operations. The best technique, with some sort of magic value passing, would therefore allow all ALU instructions in a trace line to fuse together into a single fusion instruction.

These two baselines will serve as the main comparison for the fusion techniques to showcase the absolute best that could ever be done under any circumstances. They will highlight how close this work, some of the first of its kind, is to these perfect cases. It is important to note, however, that these two "perfect" cases are not realistic, and represent an unobtainable goal.

3.6 Required Architecture

There are three main additions to the standard processor which must be added to accommodate the Dynamic Instruction Fusion architecture. The first is a smaller, more efficient register file which will provide the energy savings that is the primary goal of this work. The second is the addition of a Fusion Selection Engine (FSE) which is, as its name implies, responsible for deciding which instructions should be fused. The final core component is an enhanced ALU. This Fusion ALU must be able to handle the fused instruction streams which will be issuing through the pipeline.

In order to provide an energy savings there must be some sort of change to the processor to allow us to not have to read / write every register of all in-flight instructions. This can be done one of two ways. The first is to instrument a complicated forwarding logic system, or pipelined execution engine which allows the register values to pass from one stage to another. This approach has the benefit of not requiring any intermediate storage of register values, but does cost more in the overhead of the system. This approach is similar to that described by Bracy et al. [6] which takes advantage of a pipelined ALU with the ability to write out the results from any stage of the ALU pipeline.

An alternative approach is to use a small and energy efficient register file (Transient Register File) tightly coupled to the Fusion ALU in order to provide the savings. Transient register values are stored in this structure in order to allow them to be consumed by the Fusion ALU. This method takes advantage of the fact that small memory structures, with fewer ports, are much less expensive energy-wise. By “caching” the transient register values, the expensive accesses to the large, many ported register file are avoided in favor of the smaller coupled structure. Sizing this register file is an important design consideration, as it needs to be able to hold all of the unique registers that will be needed by fused instruction. Experiments, detailed in Chapter 4.6 Register File Sizing, prove that sizing this register file around 8 entries (32 – 128 bytes) provides greater than 99% of the coverage of all fused instructions, with a mere 4 entries required in order to provide 91% coverage. Compared to the typical register file at ~160 or more entries, this is a substantial savings. If using the efficient register file approach, extra work must be performed to load in the initial registers required by the fused instruction, as well as writing out the resulting registers. Given the results in Chapter 4.6 Register File Sizing, this overhead cost should be very low compared to the savings achieved.

The Fusion Selection Engine contains the logic responsible for deciding how to fuse the instructions together, as well as the memory structures to hold the instructions as they are being constructed. In particular, a set of queues are needed for the each of the types of instructions which will be held, pending their being sent to the execution stage. The examination of the instruction mix shows that load and store instructions comprise 16% and 13% of the total instructions, with a whopping 74% of the instructions being ALU or branch instructions which can be fused together. The queue required for the fused instructions will be the largest, with something on the order of 8-16 entries in order to take full advantage of the generously sized fusion instructions that we wish to generate. Due to the much smaller ratio of loads and stores, their respective queues can be quite small, 2-4 entries each in order to provide complete coverage for most instruction traces. Finally, a queue is required to hold the FPU operations while the FSE is fusing the instructions.

In order to avoid generating wide instructions with many bits of data being passed around, a small table will be used to store the particular instructions within the fusion instruction which will be executed. A pointer into this table will be encoded into the fusion instruction which will be decoded in the Fusion ALU to execute the instructions required as well as referencing a smaller register file which provides the energy savings we are seeking. This arrangement mirrors that implemented by Bracy et al. [6] through the use of their Mini-Graph Table and the handles associated with each entry into this table.

If the simplicity of a smaller more energy efficient register file is leveraged, there is relatively little that needs to be changed in the Fusion ALU. The only difference is that now the ALU will be accessing the smaller structure, rather than the global register file. It is possible to instead use a pipelined ALU such as described by Bracy et al. [6], however, due to the added complexity of such a structure, the preference remains with the transient register file.

3.7 FSE Example

Table 1 shows a single instruction stream taken from the execution of the SPEC 2006 mcf benchmark. Each row corresponds to an instruction within the sequence, and details the source and destination registers along with the type of instruction (ALU, FPU, Load, Store, and Branch).

Table 1: Sample Instruction Stream

#	Type		Src1	Src2		Dst1	Dst2
1	ALU		1		->	76	
2	LOAD		76		->	7	
3	ALU		1		->	1	66
4	ALU		7		->	76	
5	ALU		76		->	66	
6	ALU				->	76	
7	ALU		66	76	->	76	
8	BRANCH		76		->		
9	ALU		1	12	->	76	
10	ALU		76		->	66	
11	ALU				->	76	
12	ALU		66	76	->	76	
13	BRANCH		76		->		
14	ALU		7		->	76	
15	ALU		76		->	4	
16	ALU		4		->	76	

Figure 3 shows the same instruction sequence in graphical form, and allows for an easy way to visualize the flow of data through the sequence. Each of the green lines connects the source to the destination registers for an ALU operation. The blue lines correspond to Load operations, while orange lines mark the Branch instructions which do not produce a value to be written into a register (symbolized by the empty box). This stream is organized to show the ordering of the dependencies between the instructions, and to visually demonstrate that there are actually several places where the stream is inefficient in the number of instructions issuing through the pipeline.

Figure 4 shows what this sequence could be compressed from 16 down to 6 instructions while still maintaining all of the required dependency information. This is one form of instruction fusion. It is possible to imagine this sequence being fused in such a way that each of the 6 rows of instructions could be executed together as there are no dependencies among the instructions on a row. It is also important to note that there is no way to compress the sequence into fewer than 6 instruction cycles to execute, due to the length of the longest path through this stream requiring 6 cycles (R1 → R76 → R7 → R76 → R66 →R76 →BR).

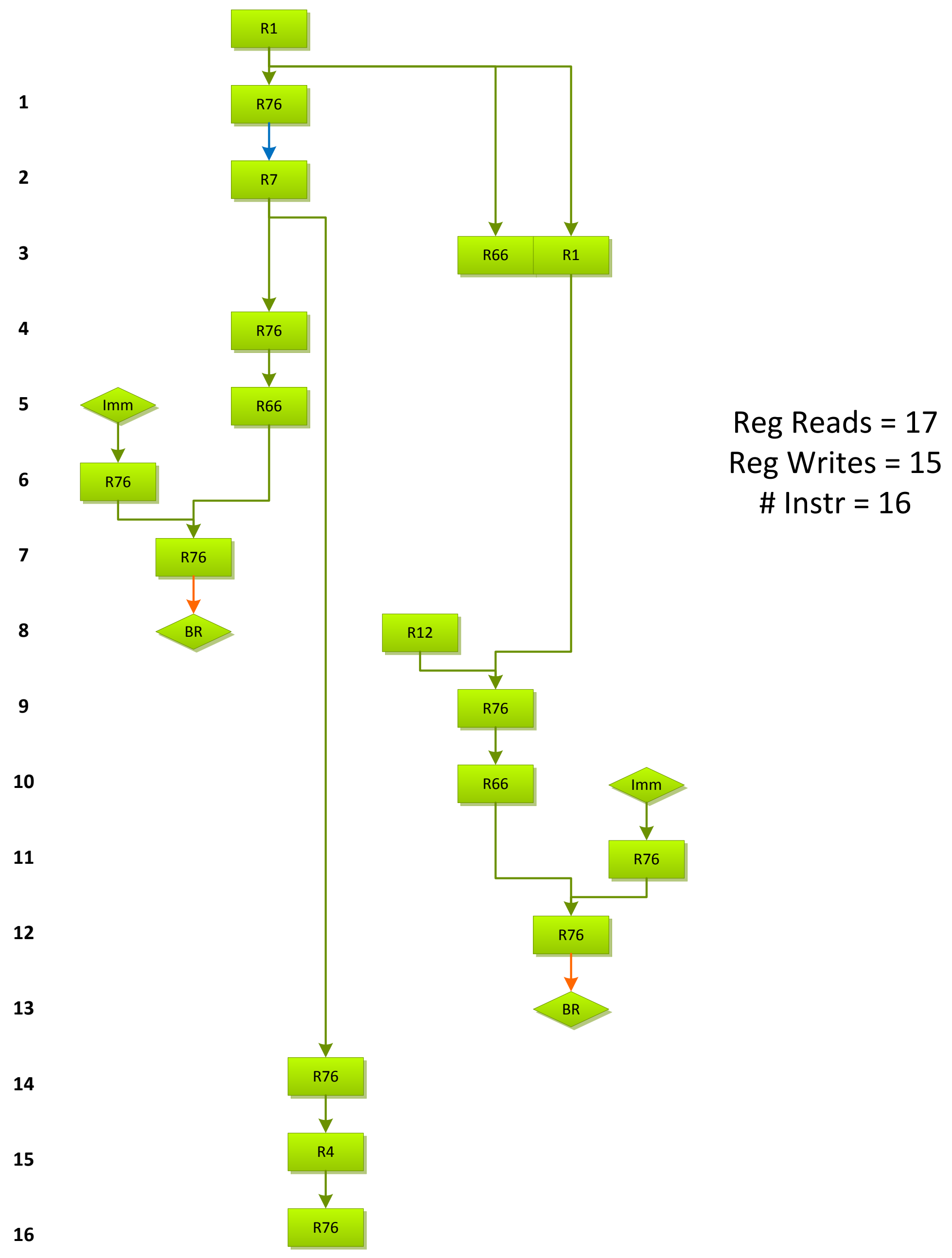

Figure 3: Sample Instruction Stream Visualization

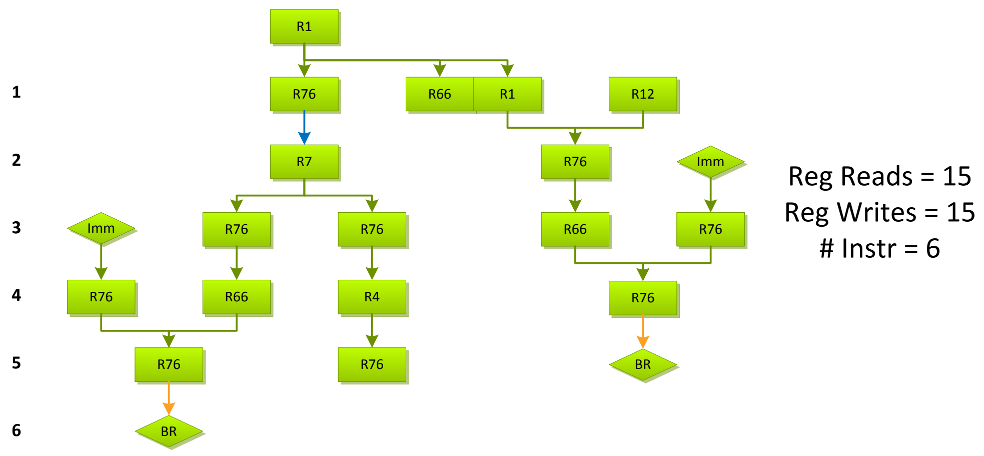

Figure 4: Sample Instruction Stream with Compression

By applying the Queued Dynamic Instruction Fusion technique described in Chapter 3.4 Queued Fusion, we obtain the fusion shown in Figure 5 and Figure 6. This fused instruction stream queues the Load (R76 → R7), allowing the instruction following it (R1 → R66 + R1) to fuse with the first instruction and eliminate a read from the execution. This same savings is also provided with the compression shown in Figure 4. The Queued Dynamic Instruction Fusion adds an additional savings on top of this. After the load is completed, all of the subsequent instructions are ALU operations which can be fused into a single fused instruction which requires reading in registers R1, R7, and R12 and produces values for R4, R66, and R76. In reality R66 & R76 correspond to the temporary register used for load/store addresses as well as branching instructions, and therefore are likely not to be needed after the fusion instruction stream completes executing. However, in order to guarantee the correctness of the register values after the fusion, these values are written out and therefore available to future instructions.

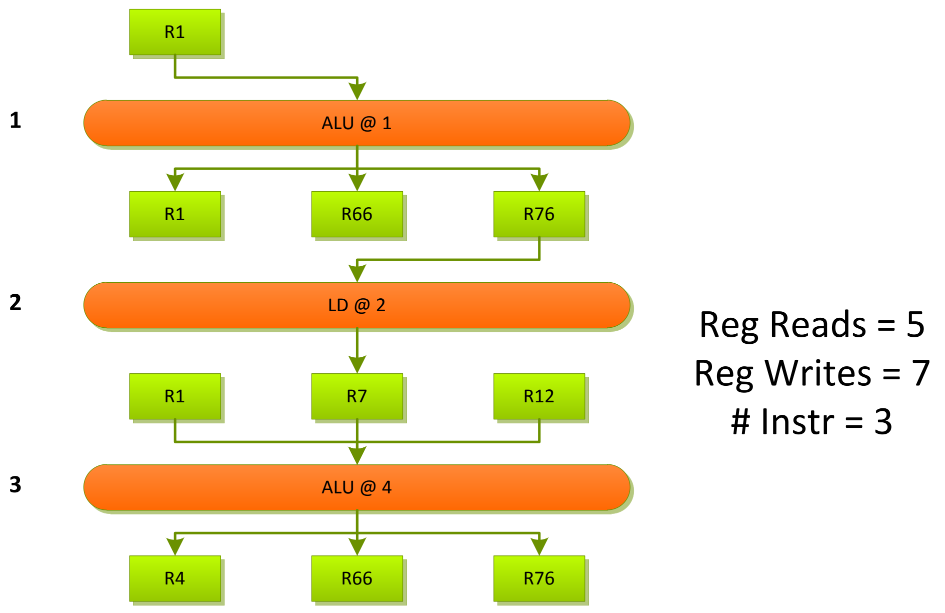

Figure 5: Sample Instruction Stream with Fusion Overview

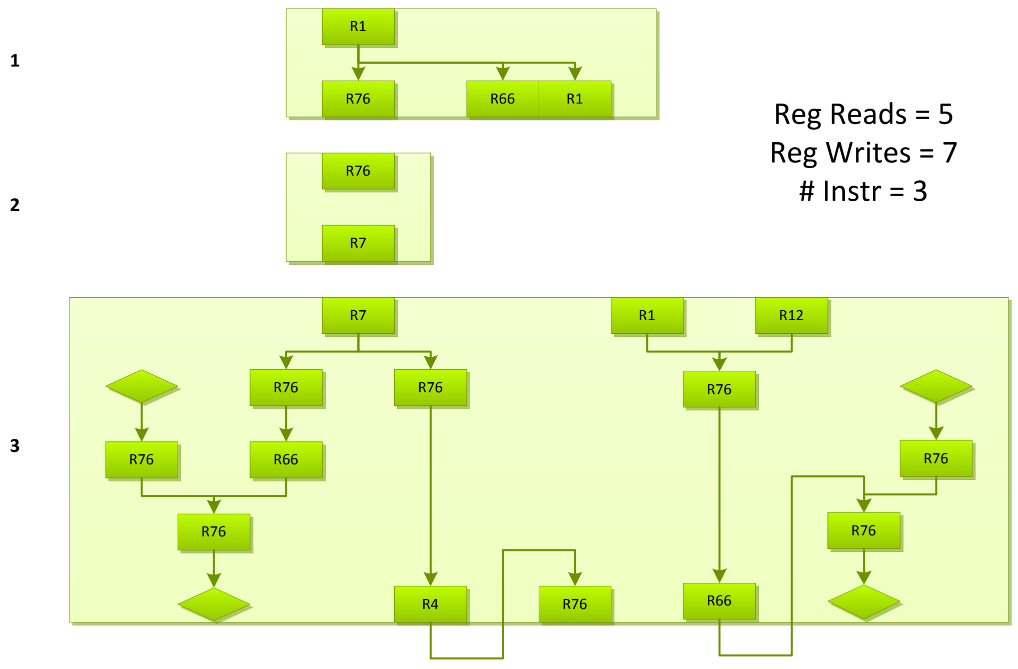

Figure 6: Sample Instruction Stream with Fusion Detailed

3.8 Further Optimizations

By examining the sequence found in Figure 3 and Figure 4, you may notice there is another option available of how to divide up the instructions into the fusion instructions. If we treat the sequence the way that it is visualized, as a graph, then we can see that there is an alternative approach to cutting the graph around the load. ALU operations connected above and to the right of the load can all be fused into a single ALU operation, while those following the load can form a second fused instruction.

In hardware, this sort of an approach would be somewhat costly. It would require either the entire instruction stream be examined as a whole, or alternatively you could allow multiple pending fusion instructions to accomplish this more complicated fusion selection. Such a modification would require that there be a mechanism to fuse pending fusion instructions together if an overlap in the dependencies was encountered. As you can imagine this would increase the complexity of the FSE design substantially.

An interesting alternative which is not fully explored in this work would be to implement these ideas and scheduling preferences into the compiler. It would be a simple matter for the compiler to detect that there were these three distinct clusters in the instruction stream graph and to optimize the clustering of as many dependent instructions together as densely as possible, in order to increase the efficiency of the in-order Fusion Selection Engine.

3.9 Limitations

Several limitations and concerns must be taken into account when designing the Fusion Selection Engine. These limitations include: the number of immediate values which are needed by a fusion instruction and the number of ports required to keep up with the computation of the enhanced ALU.

Immediate values are constant values encoded into the instructions at compile time. Unfortunately, there are a limited number of bits available to each instruction, and therefore there is no way that many immediate values could be bundled into a single instruction encoding. One way to handle the issue of dealing with multiple immediate values which might need to be encoded into a single fusion instruction is to pre-store any immediate values beyond the first into registers. In this way the FSE will be able to treat them as normal ALU operations and be none the wiser.

This of course has a penalty in the form of increased numbers of instructions issuing through the pipeline, as well as adding, rather than removing, register file accesses and total register use. Such an optimization could be provided either by a Fusion aware compiler, or handled at runtime by dynamically cracking the instructions into a pair of dependent instructions. Results show that 86% of the instruction sequences that enter the FSE contain 0, 1, or 2 immediate values. This small number of immediate dependencies is probably reasonable enough to allow for an extension to the ISA to support two immediate values per instruction; especially given that almost 50% of the streams examined have two immediate values per sequence.

The second issue to overcome is in the form of a “knob” which we can use to tweak the performance of the FSE. In particular, the number of ports into the register file as well as the ALU pipeline will have an impact on the performance that can be sustained. In addition, adding extra ports to each of these structures will result in extra energy consumption which will begin to undermine the energy savings provided by Dynamic Instruction Fusion. The effect of varying these parameters is discussed in detail in Chapter 4.4 Number of Immediate Values per Stream.

Chapter 4 – Results

4.1 Setup

In order to evaluate the performance of the Dynamic Instruction Fusion technique, the suite of SPEC 2006 benchmarks are run through the ESESC architectural simulator. Each benchmark was sampled using the SMARTSmode sampling method [11] [12]. This method has four modes: Rabbit, Warm-Up, Detail and Timing as described in Table 2: ESESC Simulation Modes. Each benchmark was run for a maximum of ten billion instructions, with an average of 2.3 billion instructions going through the full Timing simulation.

Table 2: ESESC Simulation Modes

Phase	Description
Rabbit	Fast-forward emulation or native co-execution
Warm-up	Memory and branch traces to maintain accurate state
Detail	Cycle-accurate modeling, statistics are discarded
Timing	Cycle-accurate timing modeling

Two architectural designs are evaluated for this work. The first (Traditional) is a traditional modern 4-way superscalar processor, capable of fetching four instructions and providing a single branch prediction each cycle. The second (Fusion) is a fusion architecture designed to match the functionality that would be provided by either a trace cache or a wide issue processor. Namely this architecture is capable of fetching 16 instructions and producing three branch predictions per cycle. The results in each section below will be provided for both of these architectures in order to demonstrate the effectiveness of the Dynamic Instruction Fusion technique on both a conservative and a more aggressive architecture.

4.2 Instruction Stream Sizing

Modern branch predictors are extremely good at predicting the paths of branches. Therefore the number of instructions available to the Fusion Engine is quite high, with 90 % of the instruction streams on the Traditional 4-way superscalar architecture containing the maximum of 4 instructions (Figure 7). Similarly, on the Fusion architecture, over 60% of streams consist of the full complement of 16 instructions each (Figure 8).

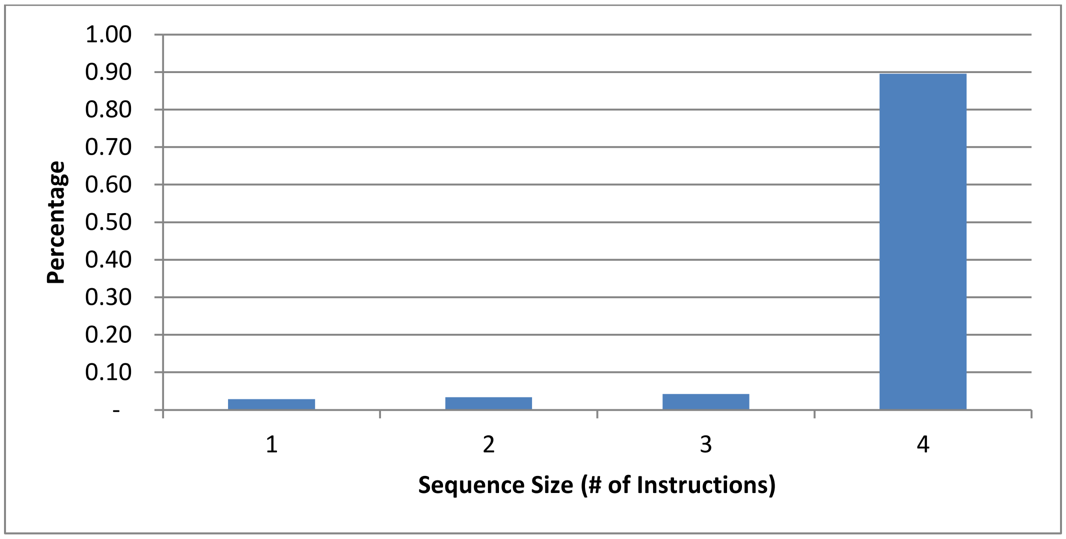

Figure 7: Average # of Instructions per Instruction Stream on Traditional Architecture Across all Benchmarks

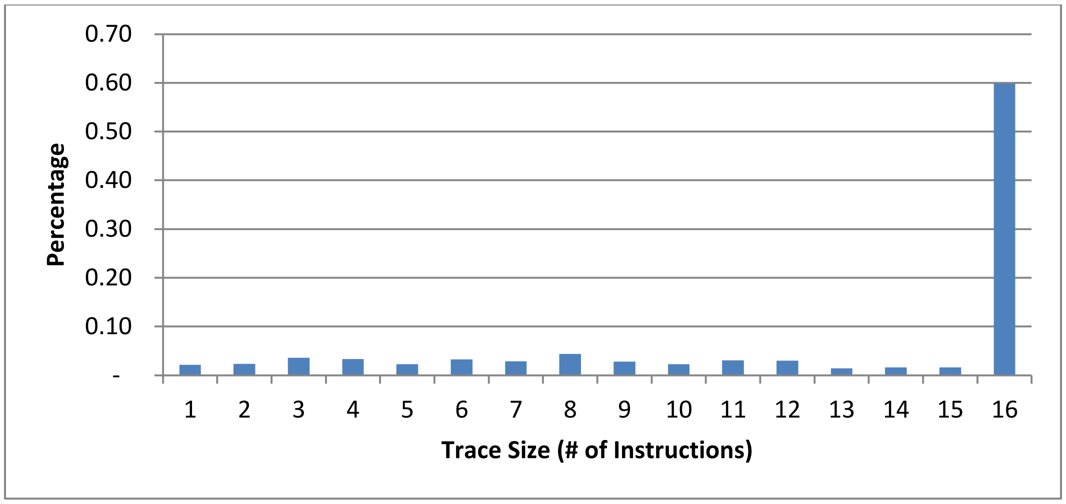

Figure 8: Average # of Instructions per Instruction Stream on Fuse Architecture Across all Benchmarks

4.3 Reductions – Register Accesses and Instructions

Two primary metrics are used to evaluate the performance of the Dynamic Instruction Fusion technique: register file accesses and the number of instruction handles issued through the execution pipeline. Here the results are reported for the Naïve (Chapter 3.3 Naïve In-Order Fusion) and Queued (Chapter 3.4 Queued Fusion) fusion algorithms as well as the Perfect or Unique results (Chapter 3.5 "Perfect" Fusion). All results are reported relative to the baseline run where no fusion is performed, and the instructions issue normally (sequentially and independently).

On the traditional architecture (Figure 9 and Figure 10), with at most four instructions available for fusion, the Naïve and Queued fusion algorithms provide a 17% and 21% reduction respectively in the number of register file accesses, while providing a 38% and 44% reduction to the number of instructions issued to the execution engine. These results are reasonably good in comparison to the

“Perfect” results which would provide a 37% reduction to register file accesses as well as a 48% reduction to the number of instructions issued, despite having only a few instructions available to perform the fusion on.

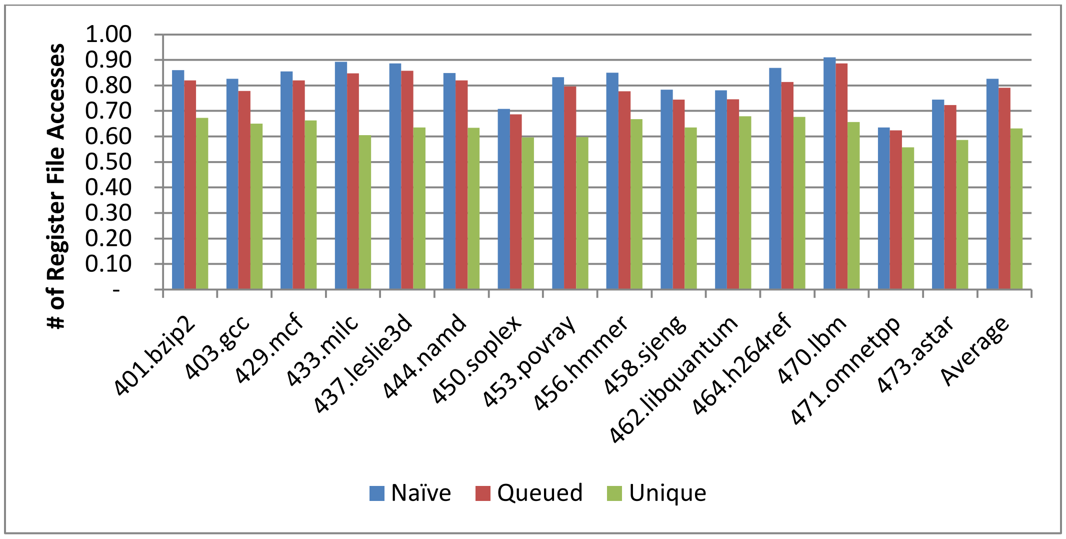

Figure 9: Traditional Architecture -- # of Register File Accesses, Normalized to Non-Fused Baseline

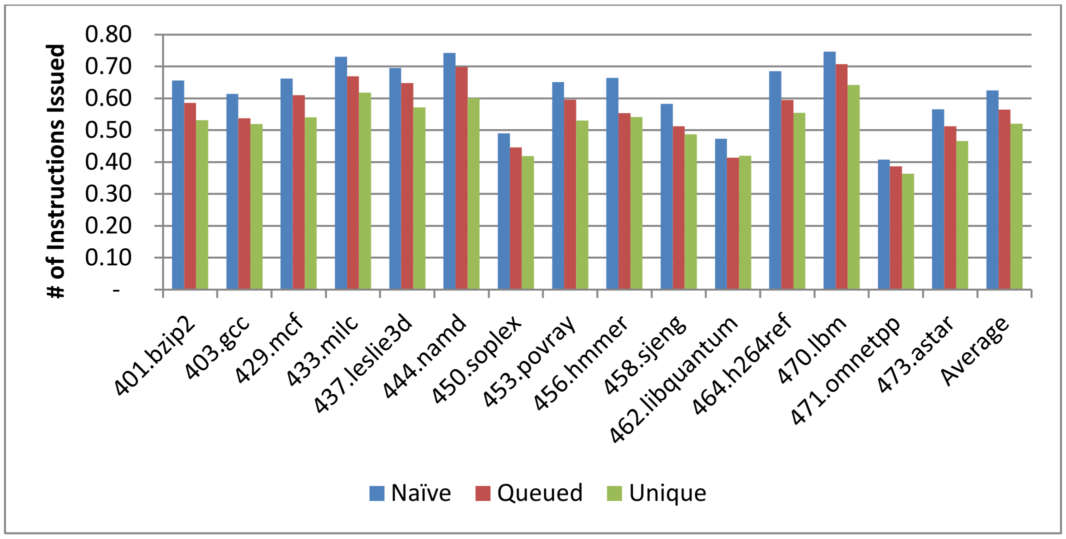

Figure 10: Traditional Architecture -- # of Instructions Issued, Normalized to Non-Fused Baseline

Slightly better results are achieved when the number of instructions evaluated by the FSE expanded to 16 instructions on the Fusion architecture (Figure 11 & Figure 12). Here we see that there is an available 24% or 31% reduction to the number of register file accesses using the Naïve and Queued fusion algorithms respectively. An even more drastic an improvement is provided to the number of instructions issued seeing reductions of 45% (Naïve Fusion) and 54% (Queued Fusion). Amazingly, if it were possible to obtain the results provided through the "Perfect" fusion (which, as a reminder, is a physical impossibility due to instruction dependencies) one would see an astounding 64% and a 65% reduction to the number of register file accesses and instructions issued respectively.

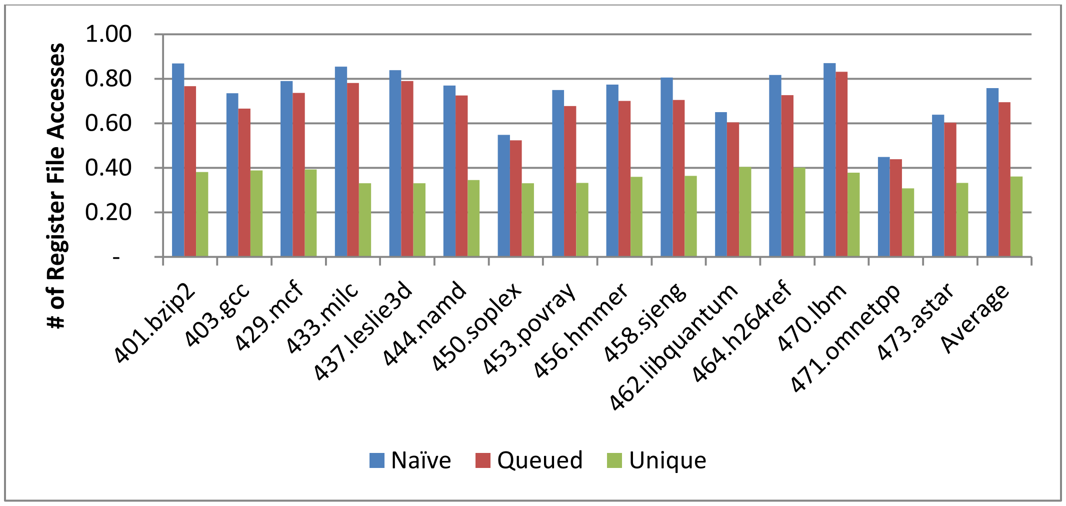

Figure 11: Fusion Architecture -- # of Register File Accesses, Normalized to Non-Fused Baseline

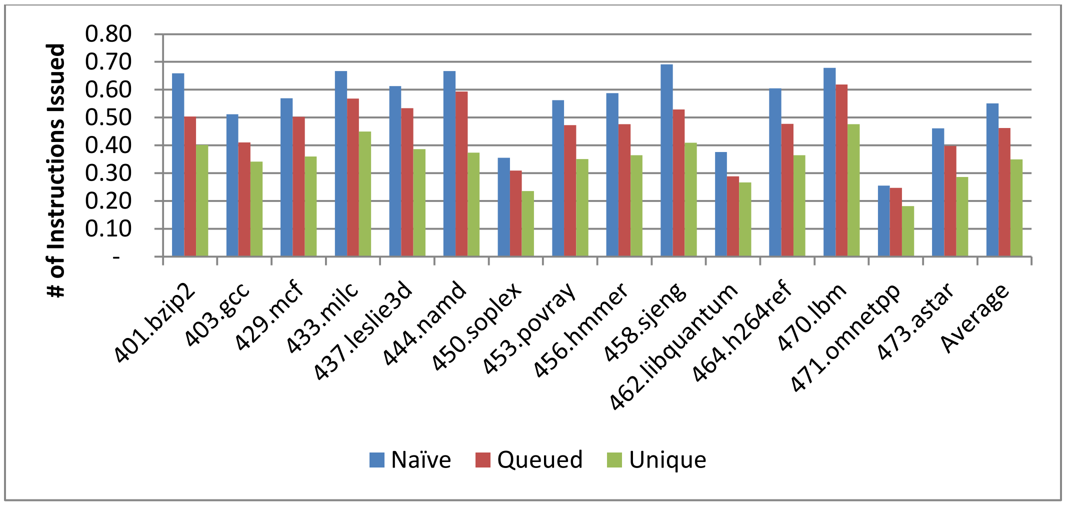

Figure 12: Fusion Architecture -- # of Instructions Issued, Normalized to Non-Fused Baseline

4.4 Number of Immediate Values per Stream

Immediate values are common to many types of instructions in modern processors. In branching instructions, the offset to the new code is a fixed distance away in memory, and therefore the offset from the branch instruction to the target of the branch is known at compile time. This value is encoded into the instruction, and is retrieved at decode time. By examining the instruction streams in the SPEC Benchmarks, it is revealed that a large percentage of instructions require an immediate value be encoded into the instruction. When the scope of this examination is broadened to encompass instruction streams consisting of four instructions for the Traditional architecture, and 16 instructions on the Fusion architecture, it is clear that the vast maturity of these (62% and 91% respectively) require more than a single immediate value per fused instruction. Figure 13 and Figure 14 are normalized histograms which provide insight into the frequency of immediate values within the instruction mix.

Constrained by the limited number of instructions evaluated by the FSE on the Traditional architecture, a full 86% of the incoming instruction streams contain at most two immediate values. This small number of immediate values is something that could conceivably be built into a new Instruction Set Architecture at the cost of the extra bits required to store the instruction. In the case of the Fusion architecture, an average 91% of the instruction streams contain at most eight immediate values. This number is unreasonably large for it to be considered being supported in the ISA, and an alternative method must handle this many immediate values.

One possible way to handle this case is to pad out the beginning of the instruction stream with instructions which are capable of pre-loading the immediate values into temporary registers to be consumed by the fused instruction. This approach has the drawback of cutting into the savings by the FSE in terms of the number of register file accesses. This penalty could potentially be offset by the ability to "re-use" some of these immediate values. It is reasonable to assume that only a handle of values are going to be used by the instruction stream, and that you could re-use these values for multiple instructions by loading them into a temporary register. In particular, the values 0 and 1 are very common immediate values to observe encoded into the instructions.

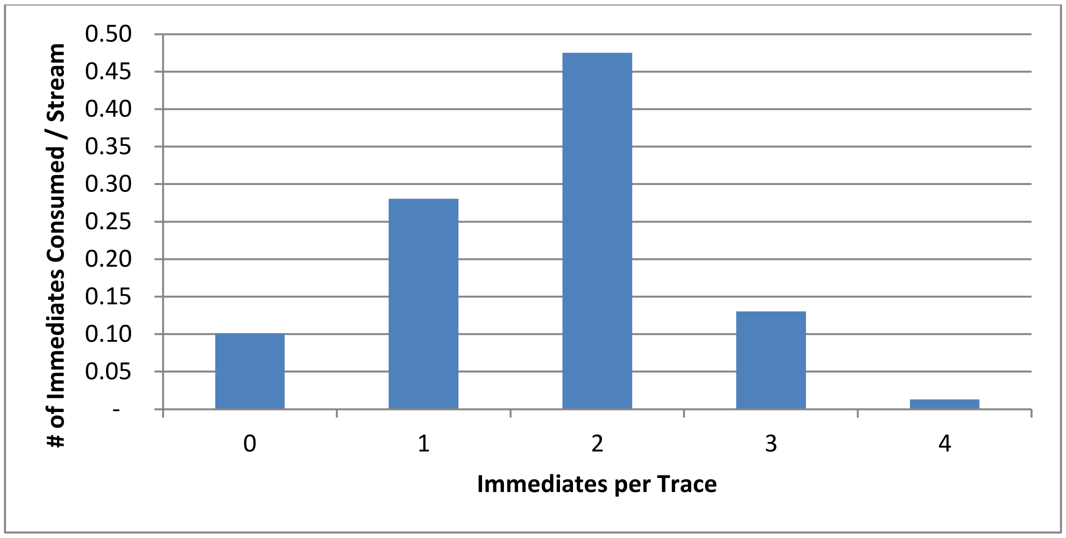

Figure 13: Traditional Architecture -- # of Immediates Consumed per Instruction Stream

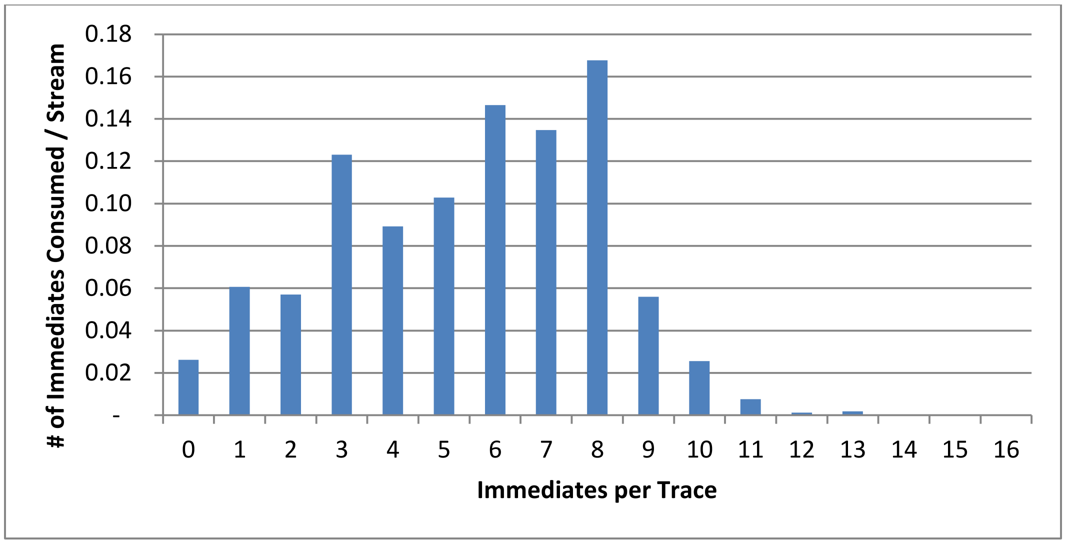

Figure 14: Fusion Architecture -- # of Immediates Consumed per Instruction Stream

4.5 CACTI Simulation Results

In order to determine the energy consumption, and in particular the savings provided by the Dynamic Instruction Fusion technique, it is necessary to model the physical register file structure. To accomplish this, CACTI 6.0 is selected for its ability to model large cache structures [13]. Four main focuses were examined in CACTI to evaluate the sizing of the of the memory structures to be used. Figures Figure 15, Figure 16, Figure 17, and Figure 18 show the CACTI results graphically for access time, area, read energy, and leakage power respectively. Each series in each of the plots corresponds to the size in bytes of the register file modeled. The number of ports provided for the register file plays a substantial role in the performance of the structure, with more ports bringing higher costs in access time, area usage, and increased read energy and leakage power. Within each specified cluster of ports (i.e., 4 ports) the results for the various size memories are given.

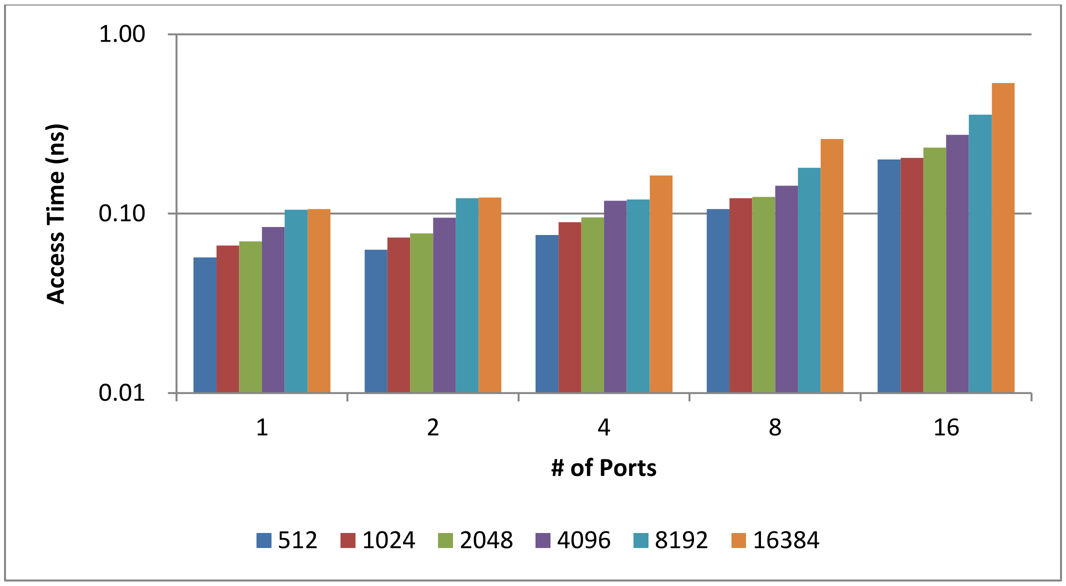

Figure 15: CACTI - Access Time by Register File Size

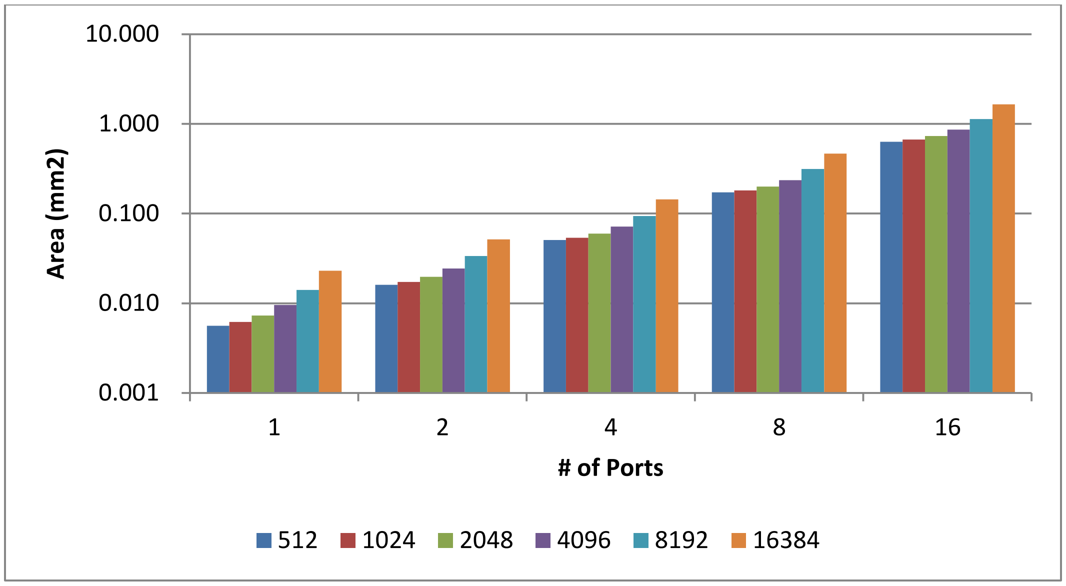

Figure 16: CACTI – Area by Register File Size

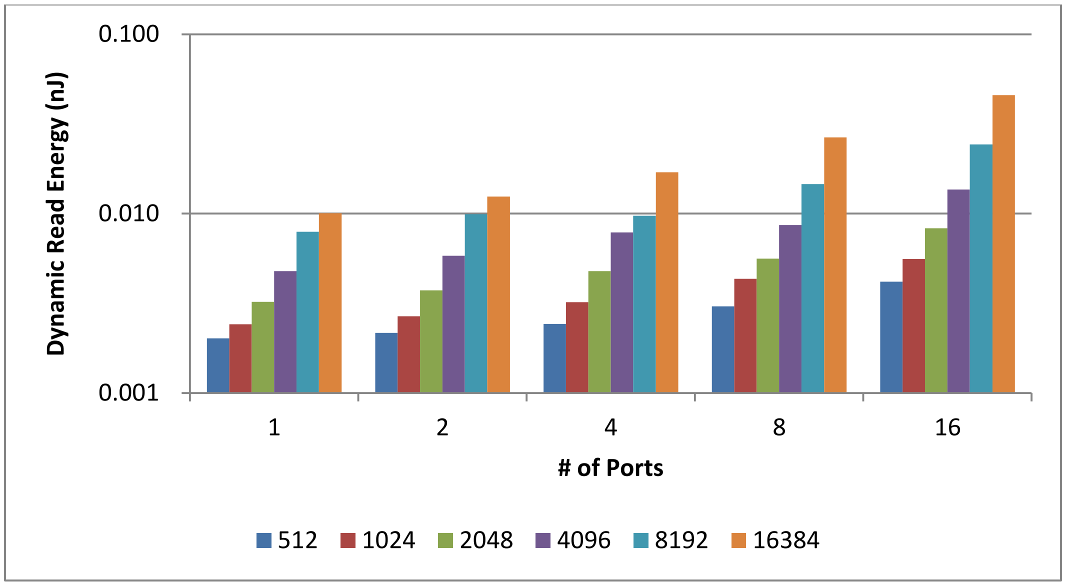

Figure 17: CACTI - Dynamic Read Energy by Register File Size

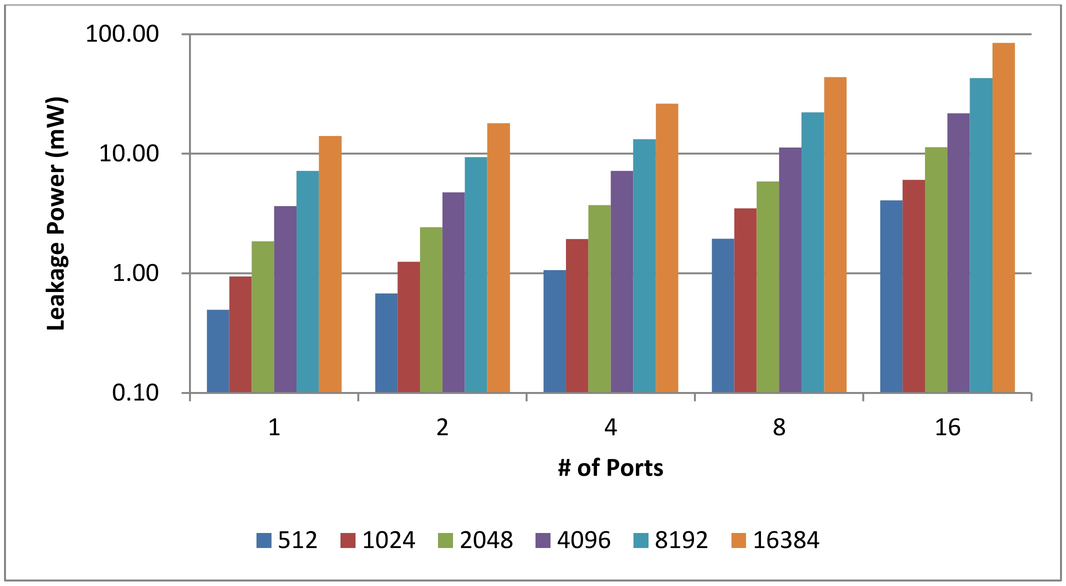

Figure 18: CACTI - Leakage Power by Register File Size

4.6 Register File Sizing

The main goal of Dynamic Instruction Fusion is to reduce the energy consumption of the processor. Fusing instructions together allows for fewer accesses to the global register file. As we have just seen, we can reduce the number of register file accesses by an average of 31% by applying the Queued fusion algorithm to sequences of 16 instructions.

In a perfect system that was able to forward the intermediate data dependencies between instructions executing through the Fusion ALU, without having to write these values to any memory, this would result in a 31% savings in the dynamic read energy consumed by the physical register file. In reality, as described in Chapter 3.6 Required Architecture, we know that a simpler system would involve a smaller register file, sized to hold only the intermediate register values for the ALU, and should produce a better efficiency. Figure 19 & Figure 20 show that constraining the fusion instructions to allow at most 8 unique registers each, provides 99.9% and 99% coverage for the Traditional and Fusion Architectures respectively. Limiting the size of the Transient Register File proposed in Chapter 3.6 Required Architecture to a small 8-entry size amounts to a TRF of between 32 and 128 Bytes depending on the size of the registers needed (32 – 128 bit). For a port count, 2 or perhaps 4 ports could be selected to balance the performance of reading and writing from the Transient Register File each cycle.

Unfortunately, CACTI is unable to simulate memory structures smaller than 512 Bytes, therefore results for a 32 or 128 Byte TRF are omitted from this thesis, but from examining the trends in the results from the CACTI simulations we can expect that there will be a 25-75% improvement to the four aspects that we are focused on: access time, area, read energy, and leakage power vs the 512 byte structure.

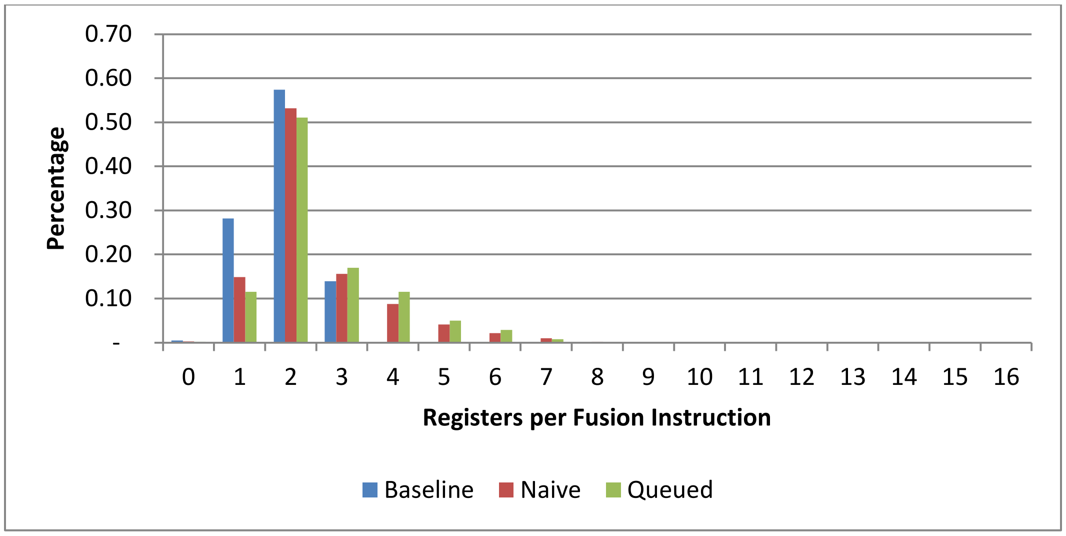

Figure 19: Traditional Architecture -- Registers Used per Fusion Instruction

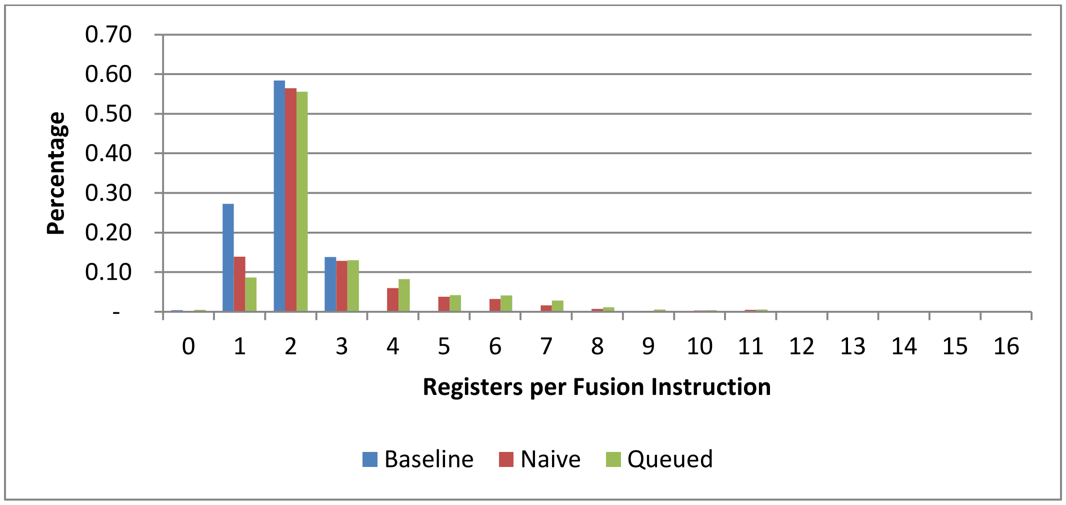

Figure 20: Fusion Architecture -- Registers Used per Fusion Instruction

4.7 Potential Architecture

Leveraging these results allows a direct insight into the energy saving potential of the Dynamic Instruction Fusion technique. For the traditional / baseline architecture we select a 16 KB, 16 port register file compared to the fusion architecture of 512B, 4 port register file. The fusion architecture sizing is chosen to optimally cover the number of registers used per instruction stream, given the results in Chapter 4.6 Register File Sizing. Namely the 512 byte TRF is chosen as it can hold eight 64-bit values, which experiments show covers 99% of all fused instruction streams.

Table 3 shows the access time, area, dynamic energy, and leakage power of both the TRF as well as the traditional baseline register file. We can see that the power overhead of the 512 byte TRF is a mere 1.06 mW, with an area of 0.05 mm2. With the access time to the TRF being approximately 7x faster than that of the baseline architecture, there should be plenty of room to accommodate the overhead of selecting to load values from the TRF rather than the global register file, when appropriate.

Table 3: Transient Register File vs Traditional Register File

	Baseline Architecture	**Fusion Architecture**	**Improvement**
Size (bytes)	16384	512	--
Ports	16	4	--
Access Time (ns)	0.53	0.08	7.03x
Area (mm^2)	1.65	0.05	32.66x
Dynamic Energy (pJ)	45.67	2.43	18.81x
Leakage Power (mW)	84.66	1.06	79.64x

The main purpose of this work is to provide a means of saving energy through the reduction of the number of accesses to the global register file. This is accomplished by instead storing intermediate values in the Transient Register File, where accesses are substantially cheaper, with a low power and area overhead. Figure 21 shows that the savings energy from leveraging the TRF is nearly the same as the savings in register file accesses overall, as shown in Figure 11. This improvement comes from the fact that the TRF accesses are ~19x cheaper than those to the global register file.

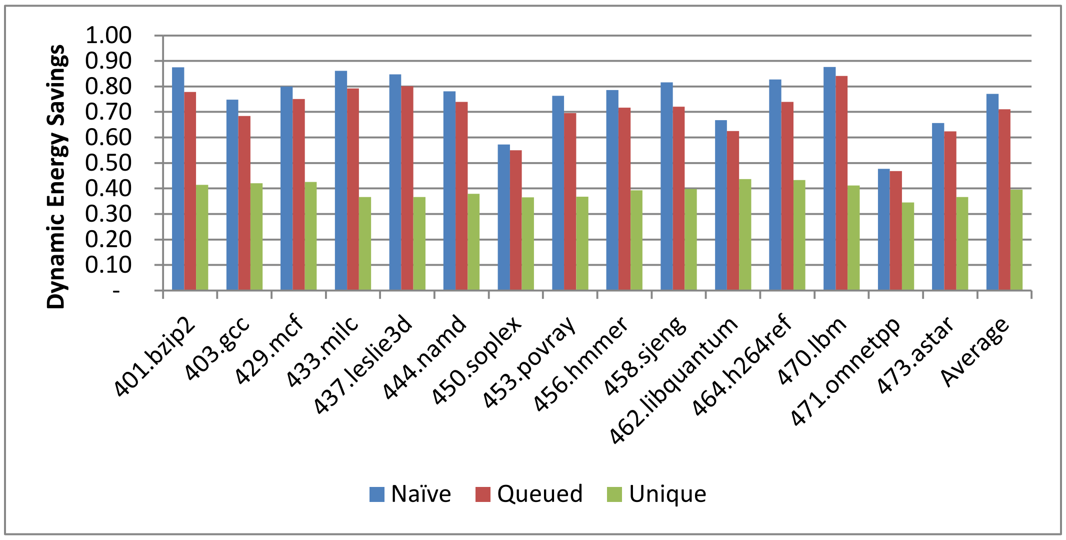

Figure 21: Savings in Dynamic Energy Consumption by leveraging the Transient Register File

Chapter 5 – Conclusions

This work introduces a simple, in-order, hardware based approach to dynamic instruction fusion. Instruction fusion provides a powerful means to reduce both the number of accesses to the register file, eliminating an average of 32% of reads and 25% of writes, while reducing the number of instructions issuing through the processor pipeline by 48%.

Bibliography

[1] O. Mutlu, J. Stark, C. Wilkerson and Y. N. Patt, "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro*, vol. 23, no. 6, pp. 20-25, 2003.

[2] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th International Symposium on Microarchitecture (MICRO)*, 1996.

[3] W. Zhang, S. Checkoway, B. Calder and D. M. Tullsen, "Dynamic Code Value Specialization Using the Trace Cache Fill Unit," in *International Conference on Computer Design (ICCD)*, 2006.

[4] D. H. Friendly, S. J. Patel and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st International Symposium on Microarchitecture (MICRO)*, 1998.

[5] A. Seznec, "Genesis of the O-GEHL branch predictor," *Journal of Instruction-Level Parallelism*, vol. 7, pp. 1-12, 2005.

[6] A. Bracy, P. Prahlad and A. Roth, "Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[7] I. Kim and M. H. Lipasti, "Macro-op Scheduling: Relaxing Scheduling Loop Constraints," in *Proceedings of the 36th International Symposium on*

Microarchitecture (MICRO), 2003.

[8] S. Vassiliadis, J. Phillips and B. Blaner, "Interlock Collapsing ALU's," *IEEE Transactions on Computers,* vol. 42, no. 7, pp. 825-839, 1993.

[9] L. Tran, N. Nelson, F. Ngai, S. Dropsho and M. Huang, "Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing," in *International Symposium on Performance Analysis of Systems and Software*, 2004.

[10] P. G. Sassone, D. S. Wills and G. H. Loh, "Static Strands: Safelu Collapsing Dependence Chains for Increasing Embedded Power Efficiency," in *LCTES*, 2005.

[11] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proceedings of the 30th Annual Internation Symposium on Computer Architecture (ISCA)*, 2003.

[12] E. K. Ardestani, E. Ebrahimi, G. Southern and J. Renau, "Thermal-Aware Sampling in Architectural Simulation," in *International Symposium on Low Power Electronics and Design*, Redondo Beach, California, 2012.

[13] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," International Symposium on Microarchitecture, Chicago, 2009.

[14] I. Kim and M. H. Lipasti, "Implementing Optimizations at Decode Time," in

Proceedings of the 29th International Symposium on Computer Architecture (ISCA), 2002.

[15] A. Moshovos and G. S. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," in *Proceedings of the 30th International Symposium on Microarchitecture (MICRO),* 1997.

[16] A. Marquez, K. B. Theobald, X. Tang and G. R. Gao, *A Superstrand Architecture*, 1997.

[17] P. G. Sassone and D. S. Wills, "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, 2004.