Dynamic Dependency Collapsing

By

Görkem Aşılıoğlu

A Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In Computer Science

Michigan Technological University

2020

This dissertation has been approved in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science.

Department of Computer Science

Dissertation Advisor: *Soner Önder*

Committee Member: *Zhenlin Wang*

Committee Member: *Saeid Nooshabadi*

Committee Member: *David Whalley*

Department Chair: *Linda Ott*

For my parents,

who have been my endless source of support.

Contents

List of Figures

List of Tables

Author Contribution Statement

Parts of the material contained in Chapter 3 was previously published in Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15) [2]. I am the first author of the published article, and contributed the design ideas, designed the experiments and simulators, and evaluated the results for the work.

Material contained in Chapter 4 is planned to be published in a conference. I am the first author of the draft manuscript and contributed the design ideas, designed the experiments and simulators, and evaluated the results of the work.

Acknowledgements

This work would not be possible without the help and assistance of the people listed below.

My family My parents, Gülten and Kasım Aşılıoğlu, have been an endless source of support throughout my studies. I am eternally thankful for their unfailing support and their patience.

My advisor Dr. Soner Önder's endless patience in helping me through roadblocks in my research and guidance in general has been invaluable both in the creation of this work as well as the rest of my academic career. This work would not have been possible without his support and encouragement.

A heartfelt thanks also goes to all my friends, peers and collegues throughout Michigan Tech who have supported me in the creation of this work.

Abstract

In this dissertation, we explore the concept of *dynamic dependency collapsing*. Performance increases in computer architecture are always introduced by exploiting additional parallelism when the clock speed is fixed. We show that further improvements are possible even when the *available parallelism* in programs are exhausted. This performance improvement is possible due to executing instructions in parallel that would ordinarily have been serialized. We call this concept *dependency collapsing*. We explore existing techniques that exploit parallelism and show which of them fall under the umbrella of dependency collapsing. We then introduce two dependency collapsing techniques of our own. The first technique collapses data dependencies by executing two normally dependent instructions together by *fusing* them. We show that exploiting the additional parallelism generated by collapsing these dependencies results in a performance increase. Our second technique collapses *resource dependencies* to execute instructions that would normally have been serialized due to resource constraints in the processor. We show that it is possible to take advantage of larger in-processor structures while avoiding the power and area penalty this often implies.

Chapter 1: Introduction

In computer architecture, parallelism is the keystone to creating faster processors when confronted with a fixed clock speed. Each architectural improvement that creates a time optimization allows some computation in a processor to happen at the same time as another. For instance, pipelining is one of the simplest examples of parallelism that exists in almost every modern processor.

Pipelining allows a processor to overlap different stages of execution of multiple instructions. Processors also execute multiple *independent instructions* at the same time when available. The number of independent instructions available for execution at any given point in a program's execution is called available *instruction level parallelism* (ILP). In the rest of this work, we use the term *available parallelism* to refer to available ILP.

Many modern micro-architectural techniques focus on *available parallelism* inherent in programs. Techniques which focus on available parallelism range from multi-fetch, multi-issue processors (superscalar processors) to multiple processors running different processes all related to the same application (supercomputers or high-performance computing). Inherently, whether or not task-level parallelism is

sought, the premise is the execution of independent instructions in parallel. Generally speaking, it does not matter if these independent instructions come from the same thread.

All micro-architecture techniques which focus on *available parallelism* hit a performance wall when the so called *available parallelism* is not sufficient. In such cases, program execution performance is bounded by *dependent code* which can't be readily parallelized by distributing it across multiple execution units.

This dissertation work aims to explore and exploit a different type of parallelism, which we call *dependent parallelism*. Exploiting *dependent parallelism* is accomplished by scheduling and executing dependent instructions together at the expense of potentially slowing down the processor clock. As a result, the dependency height of the program is collapsed and additional independent parallelism is exposed to be further exploited. Dependent code can be run in parallel using a multitude of techniques such as instruction fusion and value prediction. We refer to these techniques as *dependency collapsing techniques*. Within dependency collapsing techniques, we focus on techniques which are activated dynamically – that is, the techniques in question do their work while code is running on the processor.

In this dissertation, we also discuss the concept of *resource dependencies*, and articulate a technique which collapses such dependencies. Resource dependencies are encountered when the same architectural construct needs to be accessed by multiple instructions to exploit the *available parallelism*. Resource dependencies can only be

broken by introducing larger microarchitectural structures with a greater number of simultaneous access ports. Such additions to processors are not easy or often not feasible due to power, area and process node constraints. Our design aims to relax these dependencies by introducing multiple copies of processor structures with fewer access ports, which is often called *clustering*. We introduce a novel technique in which such additions can be made to the processor without introducing a significant performance penalty.

In summary, we explore existing dependency collapsing techniques and apply them dynamically to exploit dependent parallelism in this dissertation. As well, we design new and expanded dependency collapsing techniques to achieve greater dependent parallelism.

To make efficient use of these dependency collapsing techniques, we design, implement and evaluate novel micro-architecture designs or improvements to superscalar processors.

Chapter 2: Background and Domain Analysis

A computer program calculates some output for a problem based on given inputs. To this end, some operations must be completed after certain others, in a certain order. These orderings manifest themselves as *dependencies*. We define a *dependency* as any interaction between two operations of a program (e.g. an instruction or a basic block or an even larger section of code) that imposes an order on the execution of these operations. If an operation A must be executed before operation B, we represent it in a dependency graph as seen in Figure 2.1, where an arrow going from A to B indicates that B *depends* on A and therefore A must occur before B.

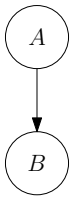


Figure 2.1: Dependency Example

As we are primarily concerned with the available *instruction-level* parallelism, in this work we focus on *instruction* dependencies instead of larger blocks of code. We

further split the concept of a dependency into *data*, *control* and *resource* dependencies.

Data dependencies occur when an instruction (B) requires a value produced by another instruction (A). In this case, A must execute first to provide the value to B for B's execution. A data dependency could occur through registers for scalar values in execution, or through memory for array values. While in theory both data dependency classes are similar, data dependencies through memory are often dynamic in nature since the memory address being accessed may vary.

Control dependencies occur when the *execution* of an instruction (B) depends on another instruction (A). Since we don't know if the instruction B will be executed at all, A must be evaluated first to determine if B shall execute. Control dependencies in a program can be converted to data dependencies using a technique called *if-conversion*.

Resource dependencies can occur when two instructions are not data or contol dependent, but instead when the execution of each instruction is dependent on the same resource in the processor. Resource dependencies impose *linearity* but do not necessarily impose a specific order. Each instruction (A or B) can be executed ahead of each other, but not at the same time due to resource limitations. A well known example of a resource dependency is a write-after-read or write-after-write register dependencies. These dependencies are created due to lack of available architectural register names in the ISA. A resource dependency can be removed by adding more of the same resource to the processor as well as the capability to utilize those resources.

For instance, a write-after-write dependency is resolved in modern superscalar processors by adding more registers, then *renaming* the architectural registers to these additional, larger bank of registers. Adding additional resources to a processor *collapses* these dependencies.

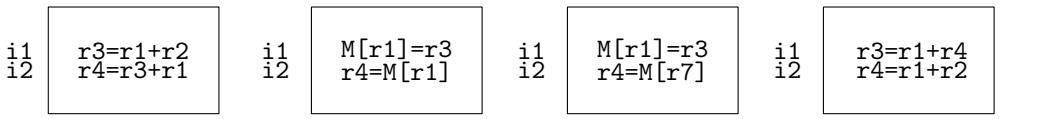
2.1 Mutability of Dependencies

We classify dependencies to be *immutable* or *mutable*. We consider a dependency to be immutable if the dependency exists in all execution paths that pass through the members of the dependency relation. Most immutable dependencies are statically determinable and mutable dependencies exhibit data dependent behavior. If two operations A and B are observed to be dependent and independent on each other at least once each, we consider the dependency between A and B to be a mutable dependency.

Consider the following code snippet shown in Figure 2.2 (a). Given this instruction ordering, $i2$ will always be dependent on $i1$ since $i1$ produces the register value $r3$ and $i2$ consumes $r3$. Immutable dependencies are also possible over memory accesses. In Figure 2.2 (b), we show that $i2$ will always be dependent on $i1$ since the value of $r1$ is not modified between the two instructions. Note that while the value storage location is not constant in this snippet, $i2$ will always read from the same location $i1$ writes to, thus forming an immutable dependency. In Figure 2.2 (c), we see another immutable dependency over memory due to $r1$ being equal to $r7$ in all paths of execution. This

dependency would not be statically determinable in most cases.

In Figure 2.2 (d), we see an immutable resource dependency. $i1$ and $i2$ are not control or data dependent. However, $i2$ may not execute ahead of $i1$ since it will overwrite $r4$, which $i1$ hasn't read yet. This dependency occurs due to the lack of available architectural registers, and exists in all paths of execution leading to this code snippet. Such dependencies are commonly known as false data dependencies. We classify them in a broader category of dependencies that all exist due to lack of resources. Additionally, any instructions that rely on the same architectural resource (such as register file ports, execution units, etc.) are immutably resource dependent as well. Regardless of the execution path or data of the program, should two instructions happen to execute in the same instruction stream and use the same resource, they will always be resource dependent.



(a) Over Registers

(b) Over Memory Statically Identifiable

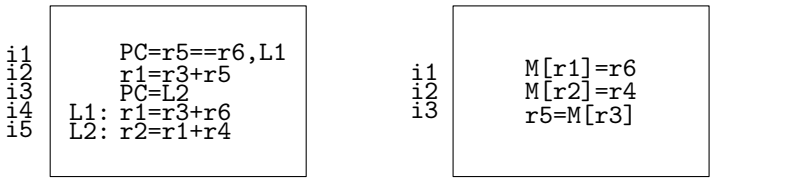
(c) Over Memory `r1==r7`, always

(d) Resource dependency Anti-dependency

Figure 2.2: Immutable Dependency Examples

Figure 2.3 (a) shows an example of a mutable dependency due to a branching control flow. If the branch at $i1$ is not taken, $i5$ would be dependent on $i2$. Should the branch at $i1$ be taken, $i5$ would then instead depend on $i4$. Mutable dependencies can also occur without branching control flows. Here, the data value tested by the branch instruction controls whether the dependency occurs. In Figure 2.3 (b), we

see a load instruction preceded by two store instructions. Based on the equality relationship between $r1$, $r2$ and $r3$, $i3$ may be mutably dependent on both $i1$ and $i2$. Again, data values control whether the dependency occurs. Resource dependencies exist between any pair of instructions in the executed instruction stream due to the accessed memory and register ports.



(a) Control Based
`r5==r6` at least once, and
`r5!=r6` at least once

(b) Data Based
`r2==r3` and `r2!=r3` at least once each, and
`r1==r3` and `r1!=r3` at least once each

Figure 2.3: Mutable Dependency Examples

2.2 Criticality

Dependencies impose a time order into program execution. Each dependent instruction must then run in a different time interval (typically a *cycle* in computer architecture). Given a dependency graph, we call the longest path through that graph the *critical path*. Consider Figure 2.4. The critical path in this graph is through $i_1 \rightarrow i_3 \rightarrow i_5 \rightarrow i_7$. To execute all instructions in this figure, we need at least 4 time intervals even when given infinite execution resources because of the dependencies.

Resource dependencies only exacerbate the situation. For instance, let's assume we are working in a micro-architecture that may only execute two instructions per time interval. The following schedule will extend the minimum execution time of the

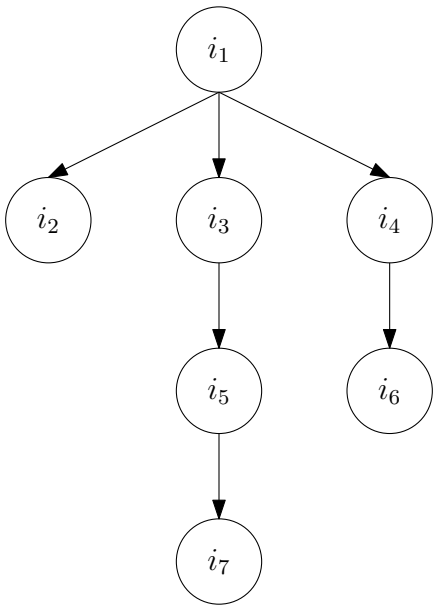


Figure 2.4: Data Dependency Example

code in Figure 2.4 to 5 time intervals: $i_1 \rightarrow i_2 \& i_4 \rightarrow i_3 \& i_6 \rightarrow i_5 \rightarrow i_7$.

Improving the performance of such a graph is possible only if the critical path is shortened.

2.3 Dependency Collapsing

Shortening the critical path can be accomplished by using *dependency collapsing*, where two or more dependent instructions are scheduled and executed together. Thus, dependency collapsing is a subcategory of instruction scheduling techniques. In addition, dependency collapsing requires techniques and mechanisms that allow multiple dependent instructions to be executed together when scheduled.

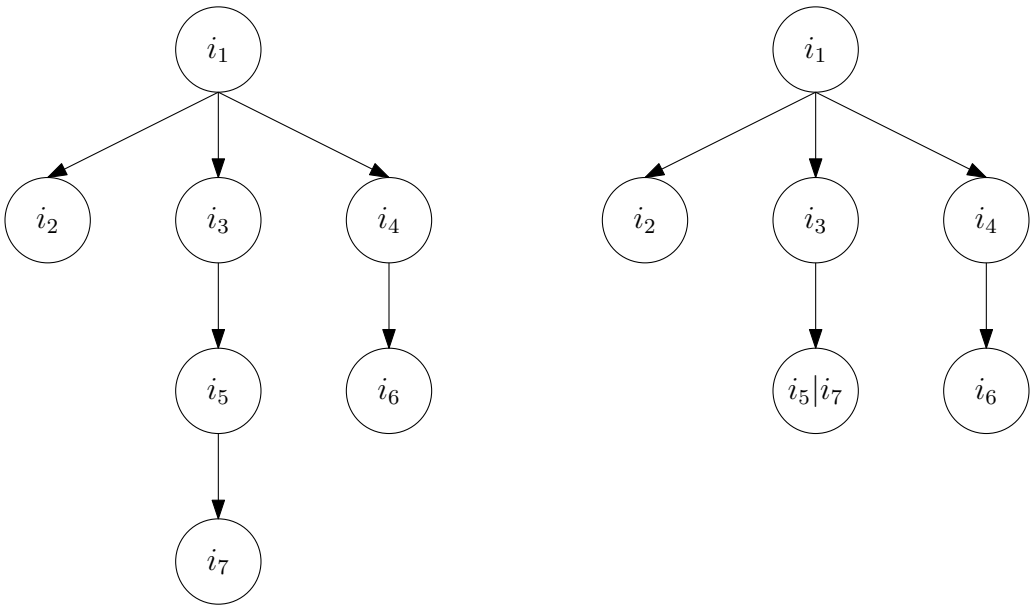
In order to take advantage of dependency collapsing to the fullest extent, it is

important to detect the critical path in the program. Critical path detection can be performed at compile time, as well as dynamically and is well studied. Critical path detection techniques range from dynamic heuristic based predictors [14, 13, 52] to instruction behavior based criticality determination [48, 23]. Heuristic predictors predict criticality based on heuristics such as how quickly an instruction's value is used. Instruction behavior based criticality determination is accomplished through identifying which instructions typically reside on the critical path, such as instructions consuming values produced by load instructions. Program trace based criticality detectors also exist [51]. However, these prior techniques assume the dependency graph does not change. Collapsing dependencies may change the dependence height of the paths containing these dependencies. Therefore, a new dependency graph may result following a dependency collapsing operation. On this new graph, the critical path may be on an entirely different branch.

2.3.1 Data Dependency Collapsing

A series of immutable dependencies between instructions is shown in Figure 2.5 (a), where i_7 is dependent on i_5. Collapsing i_5 and i_7 would yield the graph in Figure 2.5 (b). Since collapsing occurred on the *critical path*, the total height of the graph has been reduced to 3 from 4.

Note that, while collapsing on a path other than the *critical path* would still increase the available *dependent parallelism*, there would be no commensurate perfor-



(a) Original Dependency Graph

(b) Collapsed Dependency Graph

Figure 2.5: Collapsed Data Dependency Example

mance impact.

Memory Data Dependency Collapsing

While being a subset of data dependencies, dependencies through memory require additional consideration. Many memory dependencies are mutable dependencies based on data values. Consider the dashed line representing a potential memory dependency in Figure 2.6. i_7 would only be dependent on i_6 if they accessed the same memory location. Such disambiguation at compile time is difficult due to the large address space possible for each memory instruction. The logical assumption then is to consider every memory instruction to be dependent on all prior memory instructions, except that loads are independent of each other. The majority of existing dependency

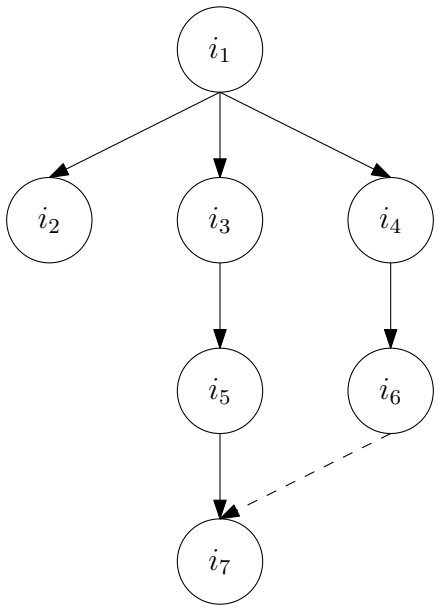


Figure 2.6: Memory Dependency Example

collapsing techniques which target memory operations begin with this assumption. Each technique then focuses on collapsing these assumed dependencies.

One well known example of a memory data dependency collapsing technique is *memory dependence prediction*[10, 34]. These techniques attempt to correctly predict whether two memory instructions are dependent on each other, instead of assuming dependence. This approach allows memory instructions that are not dependent on each other to successfully execute in parallel. If two memory instructions that were truly dependent on each other were incorrectly predicted to be independent, the processor state must be rolled back to a correct point.

2.3.2 Control Dependency Collapsing

Collapsing control dependencies enables us to schedule and execute an instruction B that is control dependent on instruction A in the same time interval. The most common control dependency collapsing mechanism is *branch prediction*, where B is executed without knowing A's result. However, once A's result is verified, B must be rolled back if a *misspeculation* occurred.

Another common technique used to handle control dependencies is *predication* or *if-conversion*. Predication makes it so that an instruction does not commit its final values to the processor state if a predicate is false. This predicate is generated by what would ordinarily be a control flow instruction. In such an instance, the control dependencies have been converted to a data dependency on a register, and may be handled according to data dependency collapsing techniques.

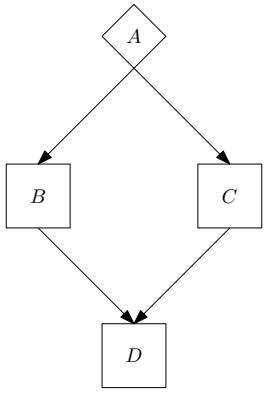


Figure 2.7: Control Dependency Example

Additionally, a processor can see control dependencies where none exist in the actual program. An example can be seen in Figure 2.7. A is a control instruction.

Depending on A's result, either B or C will be executed. However D will be executed regardless of A's result. A processor fetching several instructions at a time will see D appearing after A, and assume it is control dependent on A. There are *control independence detection* techniques [42, 9, 8, 16, 20, 1, 40] which detect instructions such as D and allow them to execute without being control dependent on A. Depending on data dependencies, collapsing may not actually allow D to execute in the same time interval as A (e.g. if D was data dependent on B or C).

2.4 Taxonomy of Dependency Collapsing

Dependency collapsing is an instruction scheduling technique. In Figure 2.8, we split instruction scheduling into two branches; *independent instruction scheduling* and *dependency collapsing*. As independent instruction scheduling is a wide and well explored field, we do not discuss independent scheduling in this dissertation. We categorize and expand on existing dependency collapsing techniques and show where they fall within our classification system.

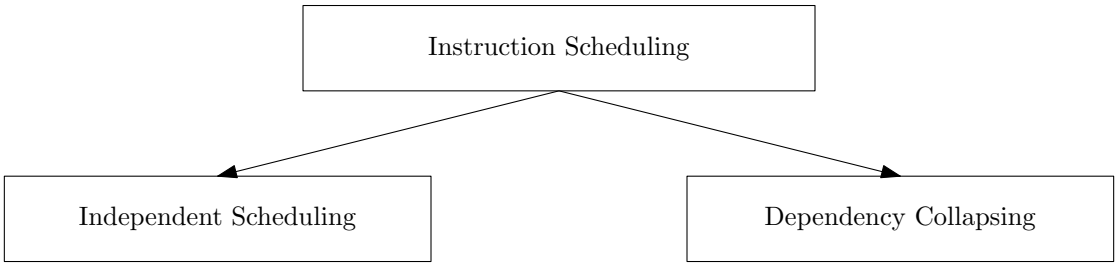


Figure 2.8: Instruction Scheduling

Figure 2.9 illustrates our classification of dependency collapsing techniques. We

identify different dependency collapsing techniques based on whether they can be applied to immutable or mutable dependencies.

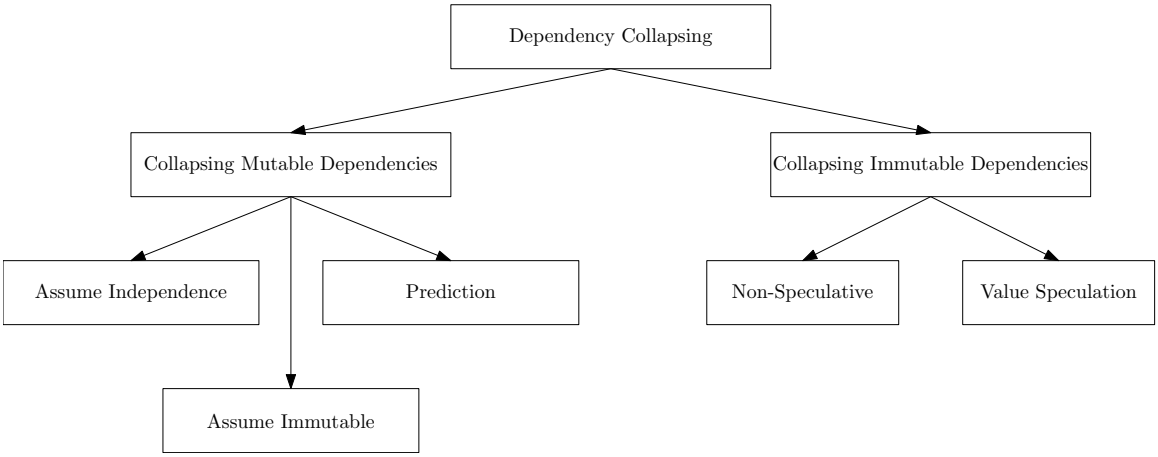


Figure 2.9: Dependency Collapsing

2.4.1 Collapsing Immutable Dependencies

Immutable dependencies do not vary over time through the program's execution. Once detected, immutable dependencies may be collapsed with any available dependency collapsing technique. Here, we classify collapsing immutable dependencies further into *non-speculative* techniques and *value prediction* techniques.

Non-Speculative Collapsing of Data Dependencies

In this dissertation, the leading non-speculative data dependency collapsing technique we focus on is instruction fusion. Instruction fusion is a technique used to execute two data dependent instructions within the same time interval through the use of a

specialized execution unit which can execute two dependent instructions in a single time interval [30].

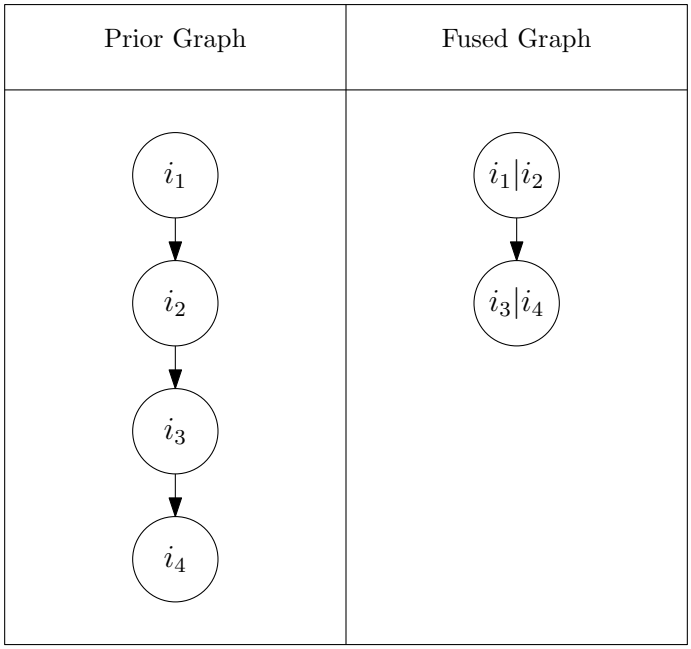


Figure 2.10: Fusion Dependency Graph

An example of fusion on a dependency graph can be seen in Figure 2.10. On the left, four dependent instructions are in a chain. Once i_1 and i_2, as well as i_3 and i_4 are fused together, the dependency chain length shortens from four to two. In other words, the execution of this graph after fusion now only takes two time intervals as opposed to four before fusion although in practice this may require lengthening the time interval. Later in the dissertation, we discuss possible hardware implementation strategies for fused instruction execution.

A fused execution unit [30] can execute two dependent instructions in a single cycle. As well, Phillips et. al. detail an example of how to build a high-performance

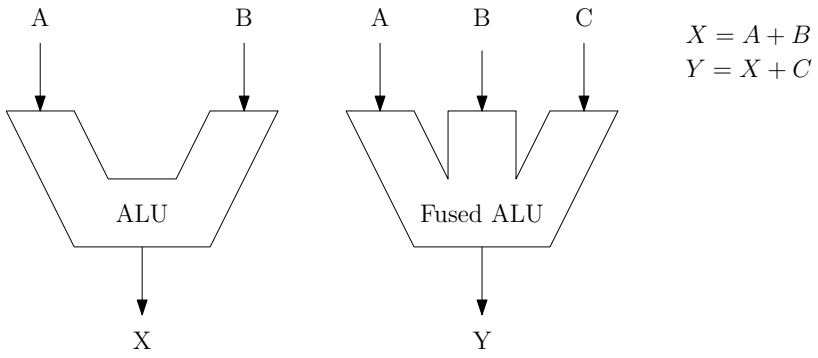


Figure 2.11: Hardware For Executing a Fused Instruction with Interim Result

3-1 ALU [39]. However, a fused execution unit only computes the final result of the fused pair. When working with techniques that dynamically fuse instructions, we often require the result of the first instruction in a fused pair as well. To produce the intermediate result as well as the final result without extending the time interval, we can use the configuration of fused and regular execution units in parallel as shown in Figure 2.11.

Instruction fusion in micro-architecture is well utilized. Fusion techniques tend to focus on two main objectives: static fusion of instructions directed toward fused execution units to achieve dependency collapsing [22, 21, 41] and dynamic fusion of instructions to achieve greater resource usage by combining instructions into smaller packages [18, 17, 25, 45, 44]. Note that many existing dynamic fusion techniques aim to improve processor resource use and not to collapse dependencies.

Non-Speculative Collapsing of Control Dependencies

Collapsing control dependencies without speculation requires an inference about the condition of the control-flow instruction. One example for non-speculative control dependency collapsing is loop vectorization. A loop condition may be pre-evaluated and if it can be shown to be true for a certain number of iterations, subsequent iterations can be run as if that branch did not exist without requiring a roll-back.

We should also note here that, although *predication* [29] is a non-speculative technique, it's not an effective dependency collapsing technique. While it does remove the branch instruction, the series of instructions that calculate the predicate result remains the same. Therefore, the predicated instructions still have the same dependency height as if they were dependent on the branch. Additional techniques may be employed for speculative collapsing of the dependencies of predicated instructions.

Non-Speculative Collapsing of Resource Dependencies

If there are resource dependencies in a program, adding additional resources will generally collapse these dependencies. However, additional processing is needed to make use of these new resources.

Resource dependencies are collapsed in three stages. First, the instructions that have the resource dependency are *translated* or *reorganized* in some way to facilitate using the added resources. In other words, the identifiers which associate a given resource with instructions that need the resource have to replaced with multiple new

identifiers, each pointing to the new resources. The instructions then make use of the additional resources to execute in parallel. Finally, the instructions are translated back to their initial state as they make their output visible architecturally, or *commit*. While there may be techniques where the first and third stages are trivial, the addition of new resources necessitates a departure from the ISA as written, which will always require a level of translation.

One example of this approach in action is the superscalar processor design itself. First, the incoming instruction stream is reorganized into a window where out-of-order instruction selection is possible. Multiple execution paths are added to the processor to make use of this out-of-order execution capability. When the instructions finish executing, they are reordered again as they make their changes to the architectural state of the processor.

Another chief example is register renaming. The registers are translated to use the name space of a much larger register file within the processor. This large register file is used during execution, which is the additional resource. Finally, the instructions *commit* their changes to the architectural state.

Value Prediction

Value prediction is a mechanism in which the result of an instruction is predicted before the instruction is complete, allowing instructions dependent on it to execute using the predicted value. As such, value prediction techniques are also an *early execution* technique. If the prediction is incorrect, these set of techniques also require

recovery action. The most common application of value prediction is load value prediction as loads typically have a high latency.

Value prediction has been introduced by Lipasti et al. [28, 27] and Gabbay [15]. It has been evaluated in hundreds of research papers as a dependency collapsing technique with various predictors. Value prediction has also been used in approximate computing.

2.4.2 Collapsing Mutable Dependencies

Mutable dependencies phase in and out of existence based on either the execution path taken to reach the dependent instruction or the difference in the input of the relevant code segment. The major challenge of collapsing mutable dependencies is discovering the existence of the dependency.

To allow dependency collapsing under uncertain dependency information, processor designs employ speculative dependency collapsing techniques. Such techniques work with two assumptions. They either assume independence between instructions and use independent scheduling techniques for collapsing dependencies, or assume an immutable dependency between instructions and use immutable dependency collapsing techniques. Many techniques in both categories keep a memoization table of dependencies where their assumptions fail, and use this table to change their assumptions as appropriate.

In addition to the default assumption of independence and immutability, predic-

tion techniques are also used to dynamically decide which assumption to use.

When a speculative assumption fails and the program order is violated, a *misspeculation* occurs. When a prediction is incorrect, it's called a *misprediction* instead. For memory dependencies, the term *memory order violation* is also commonly used. Using speculative techniques requires a state recovery mechanism. This mechanism restores the processor state to a known correct state. While some state restoration techniques are general purpose (i.e. can be used for any type of misspeculation), additional techniques can be used to speed-up or forgo state restoration for some subset of misspeculations. We briefly discuss some of the specialized techniques where appropriate.

Speculative Collapsing of Data Dependencies

Many mutable dependency collapsing techniques focus on memory instructions. Both *assuming independence* and *assuming immutability* is used for memory dependency collapsing.

One well known example of *assuming independence* is the store set algorithm [10] and its variants. The store set algorithm initially assumes no memory instruction is dependent on each other and schedules them as if they were independent. A misspeculation is detected when a store instruction that appears earlier in execution accesses the same address as a later load which executed speculatively. When a misspeculation occurs, the conflicting memory instructions are memoized in a memoization table. From that point on, all memory instructions included in the same set are as-

sumed to be dependent. Onder and Gupta improve on the misspeculation detection conditions of the store set algorithm [34] by introducing delayed misspeculation handling and value matching. In this technique, an address match between a earlier store which executed after a prior load is not sufficient to trigger a misspeculation. The value written by the store instruction must also be different from the value read by the load instruction. To ensure the latest store prior to each speculative load is the instruction that sets misspeculation status, detection of misspeculations is moved to the retire stage of the processor. This scheme allows stores to execute out of order while also stopping many false misspeculations from triggering a restart.

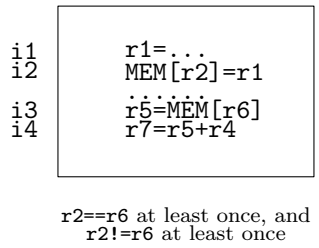


Figure 2.12: Memory Cloaking

In addition to collapsing data dependencies over memory, data dependencies can also be collapsed *through* memory when appropriate. Consider the code snippet in Figure 2.12 where if $i2$ and $i3$ are dependent on each other, $i1$ and $i4$ are also dependent on each other over register $r1$ and $r5$, despite the register names being different. We assume here that no additional changes were made to memory between $i2$ and $i3$. If the dependency between $i2$ and $i3$ can be determined, $i1$ and $i4$ could be collapsed, ignoring the memory instructions altogether. Such dependency collapsing

techniques are referred to as *memory cloaking*. Various dependency detection and cloaking algorithms have been published to achieve memory cloaking [31, 35].

Speculative Collapsing of Control Dependencies

Perhaps the most well known speculative dependency collapsing technique is branch prediction. Branch prediction collapses control dependencies by speculating whether a branch is taken or not ahead of its execution. Such a speculation allows the processor to keep fetching instructions from the predicted control path instead of having to wait for the branch to resolve. If a branch is mispredicted, the processor must undo changes made to the processor state by the speculative instructions after the mispredicted branch.

Branch prediction is an *early execution* technique. Predicting a branch completes its execution. The only thing to do after a branch is predicted is to verify the prediction, but the change in the control flow of the program has already occurred. Branch prediction has been introduced in the late 1970s. Since the early 1990s, active research has been conducted in the area beginning with Yeh and Patt's Two-level Adaptive branch predictor [54]. Currently, Seznec and Michaud's TAGE predictor [46] is considered to be the state-of-the-art. Additional research is being conducted on branch predictors using neural networks [53].

2.5 Summary

We have classified dependencies and existing dependency classification techniques in this chapter. In this dissertation, we target two major dependency collapsing mechanisms. First, we explore a technique which collapses data dependencies through the use of *fused* instructions. As well, we target resource dependencies by exploring a technique which leverages critical path extensions caused by resource dependencies into larger scale superscalar processors.

Chapter 3: LaZy Superscalar

LaZy Superscalar [1] is a novel, partially *demand-driven* micro-architecture which aims to aggressively collapse data dependencies when possible through instruction fusion. Instructions are executed when a consumer demands their result, and consumers fuse to their producers during this demand process. LaZy Superscalar achieves the task of fusing *distant* dependencies as well as data dependencies which exist across control dependencies.

While we focus on collapsing data dependencies in general, we limit ourselves to collapsing *fusible* dependencies within the LaZy Superscalar framework in this dissertation. We define fusible dependencies as two instructions which both compute a simple arithmetic operation that are data dependent. Such a limitation is necessary due to fused execution units only being capable of executing such instructions fused as described in Section 2.4.1. Therefore we do not collapse control flow, memory or pipelined arithmetic instructions such as floating point operations in LaZy

[1] Parts of the material contained in this chapter was previously published in Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15) [2],

Superscalar.

In addition to collapsing the dependency graph, a demand-driven processor also naturally *prunes* it of unnecessary instructions. Since instructions do not execute if their result is not demanded by a consumer, an instruction which has no consumers should not execute in a demand driven processor. In other words, code that is statically or dynamically dead is automatically eliminated. As a result, instructions in LaZy Superscalar have three paths to follow: (1) The instruction is executed by itself due to a demand from a non-fusible instruction (or it is non-fusible itself); (2) The instruction executes fused as part of a fused pair; (3) The instruction is discarded due to being dead.

3.1 Motivation

We approach this novel processor design by motivating the need to collapse *distant* and *mutable* dependencies. We show in Figure 3.1 that when fusion is sought, many dependent pairs of fusible instructions are only available with an interrupting control flow instruction. Figure 3.1 justifies exploring data dependency collapsing techniques that seek beyond control dependencies. On the other hand, delaying the execution of instructions until they are demanded may cause the processor to perform poorly compared to a similar processor executing instructions as they arrive due to resource constraints.

We show through a simple example in Figure 3.2 that with dependency collapsing,

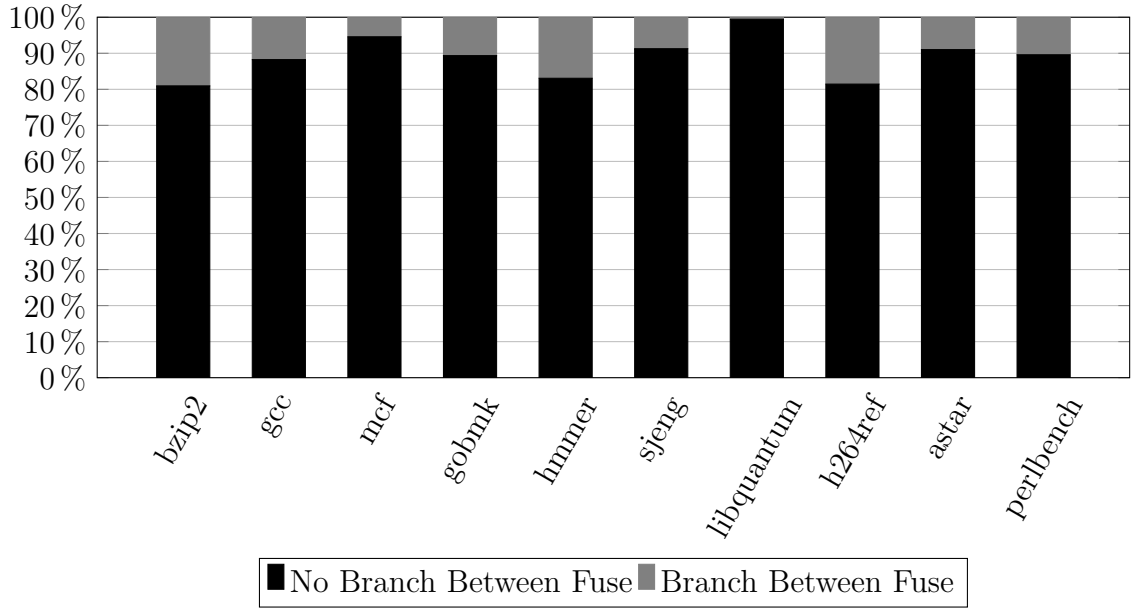
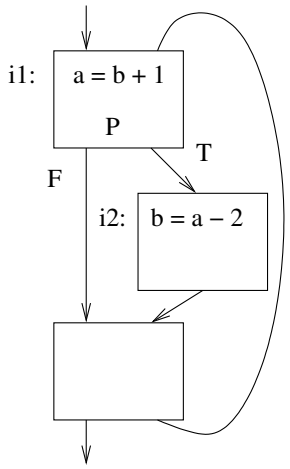


Figure 3.1: Branches Between Fusible Instructions[2]

LaZy Superscalar will at least perform no worse than an eager superscalar when execution resources are available. For this example, we only consider the instructions labeled $i1$ and $i2$ for execution, and ignore branch instructions and pipeline start-up times. We also assume the loop keeps executing. The P column indicates the predicate controlling the branch that leads to $i2$. When the P column indicates T, the branch leading to $i2$ is *taken* for that loop iteration. In the example, the subscripts for the instructions refer to the cycle in which the copy of the instruction was fetched.

The table in Figure 3.2 shows at which cycle the execution of each instruction would be complete. Note that, in the example, $i1$ is *dead* when the predicate is false, since a is not used anywhere else in the loop. For each of the six cycles of execution, an eager superscalar executes one instruction, yielding an IPC of 1. While



Cycle	P	Superscalar	LaZy Superscalar
1	T	$i1_1$	—
2	T	$i2_2$	$i1_1 + i2_2$
3	F	$i1_3$	—
4	F	$i1_4$	Eliminate $i1_3$
5	F	$i1_5$	Eliminate $i1_4$
6	T	$i2_6$	$i1_5 + i2_6$

Figure 3.2: Eager Evaluation vs Lazy Evaluation with Fusion[2]

LaZy Superscalar remains idle for four out of the six cycles of execution, it completes the execution of all necessary code at the same time as the eager superscalar, while squashing two unnecessary instructions. For this example, LaZy Superscalar can make up for time lost through delaying instructions with dependency collapsing.

In the example shown in Figure 3.2, LaZy Superscalar achieves the same performance as the eager superscalar but does not improve performance despite collapsing dependencies. This is because the example is executing and issuing one instruction at a time. Consider a case where all six instructions in the example were fetched in the same cycle. LaZy Superscalar would be able to execute two fused pairs within one cycle, completing four instructions. The eager superscalar, on the other hand, would at least need to wait another cycle to complete all instructions. This performance increase is achieved by collapsing dependencies, therefore increasing *available parallelism*. Such parallelism cannot be exploited by traditional approaches.

3.2 Demand-driven Execution

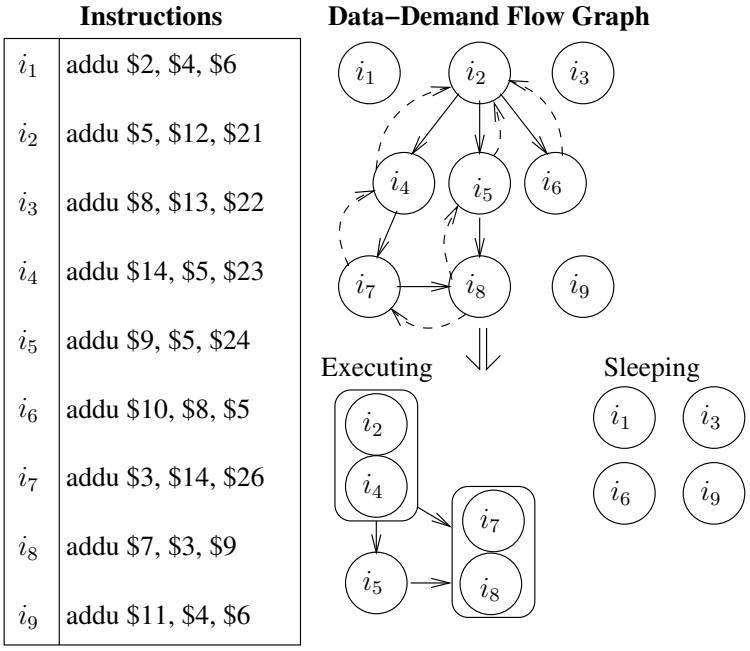


Figure 3.3: Demand-driven Execution with Fusion[2]

We briefly described demand-driven execution and scheduling in the previous sections. We now discuss the details of how consumer instructions activate their producer instructions. LaZy Superscalar accomplishes demand-driven instruction activation through *demand signals*. Figure 3.3 shows an example set of instructions scheduled using demand signals. To illustrate, we will be assuming our processor can fetch three instructions per cycle. For ease of following, each layer in the Data-Demand Flow Graph corresponds to instructions fetched within a single cycle. In the same graph, dependencies are shown with solid arrows, while demand signals are shown with dashed arrows.

When instructions i_1, i_2 and i_3 are fetched in the first cycle, all three instructions are buffered as none have been demanded yet. In the second cycle, instructions i_4, i_5 and i_6 are fetched, with all three instructions demanding i_2. i_4 is fused to i_2, and can start executing, while i_5 and i_6 are buffered. In the third cycle, i_7, i_8 and i_9 are fetched. Instruction i_7 demands i_4, but i_4 has already fused to i_2. Instruction i_8, on the other hand, demands i_7 and i_5, i_7 and i_8 fuse and are ready for execution. Note that i_5 will also be set for execution since i_8 demanded it, despite not being fused to it. Finally, instruction i_9 is independent of any instruction in our snippet, and therefore is buffered. The bottom-right graphs show the status of the processor after all three cycles have completed and all nine instructions have been fetched.

Figure 3.3 illustrates how the critical path through the program has been collapsed. The original critical path passes through $i_2 \rightarrow i_4 \rightarrow i_7 \rightarrow i_8$, resulting in a dependency height of four. After dependency collapsing, the new critical path is through $i_2|i_4 \rightarrow i_5 \rightarrow i_7|i_8$, with a dependency height of three.

Although this example clearly shows how we can track data dependencies through registers, it does not illustrate how we can manage data dependencies through memory or control dependencies. Typical superscalar processors manage these dependencies through queues. Memory instructions are directed to a load-store queue, and control dependencies are resolved at retire time, typically through a reorder buffer. To simplify and unify dependency management, in addition to assigning register names to instructions producing register results, we assign a register name to dependencies

that are normally managed by a queue. In this scheme, architectural registers are assigned physical register names as usual from a free register pool. Each name in this pool corresponds to an actual hardware storage location. Load instructions are assigned a second name aside from their register result to represent their data dependencies over memory. This name comes from a different pool of names unique to load instructions, and does not have a corresponding hardware storage location. A third name pool without storage backing is used to rename the store instructions. The final register pool is used to rename control-flow instructions, and it also does not have storage backing.

When all dependencies are "renamed" in this manner, LaZy Superscalar is able to store all the dependency information in a single bit matrix structure, detailed in Section 3.4.1.

3.3 Processor State Handling

As with any high performance processor design, LaZy Superscalar also makes use of speculation. However, when instruction execution is lazy, which means some instructions may execute very late (or not at all), determining the correct state to recover in the event of a misspeculation is a significant additional challenge. Typical superscalar processors solve this issue by enforcing in-order exit of instructions through a queue structure, commonly known as the reorder buffer. Instruction identifiers are inserted into this buffer in-order, and make final changes to processor state only when they

are leaving this buffer, also in-order. Should an instruction have a misspeculation, the final *in-order* state is well known as the misspeculated instruction is leaving the reorder buffer[2].

Reorder buffer based ordering implementations rely on the fact that instructions are *complete* when they exit the reorder buffer. With demand-driven execution, that may no longer be the case. If we attempt the naive solution and hold instructions in the reorder buffer until they complete in a demand-driven environment, we will eventually encounter a situation where the reorder buffer is blocked by an undemanded instruction. If the instruction remains until resources run out, the processor will deadlock. On the other hand, we cannot allow an incomplete instruction to leave the processor. The solution is to split the idea of state completion (*commit*) from instructions being allowed to leave the processor (*retire*). Now, instructions that are not speculative can commit their identifiers to the processor state without necessarily being complete. Instructions that are speculative must wait for their speculative status to clear to be able to commit. Commit being separate from retire allows LaZy Superscalar to store instructions in the processor until they are demanded, or the instruction is shown to be dead.

[2]There are also various implementations of the checkpointing mechanism, which takes a snapshot of the *in-order* processor state at certain intevals and restores that snapshot directly upon encountering a misspeculation.

3.4 Implementation Details

In this section, we describe the implementation details of LaZy Superscalar's pipeline. In addition, we discuss potential deadlock scenarios evident in demand-driven execution and how LaZy Superscalar avoids such scenarios.

3.4.1 LaZy Matrix Scheduler

In order to realize lazy scheduling we use a dependence matrix. Using a matrix based scheduler for conventional superscalar processors has been explored before based on the inventions by Zaidi [55] and Henstrom [19]. Our approach however differs significantly. These techniques are designed primarily for scheduling instructions, whereas our approach combines both demand and data signalling and the entire retire process is driven by the matrix based design.

The dependency matrix (DEPMAT) shown in Figure 3.4 is a single bit matrix where each line in the matrix represents the dependencies of a single instruction. A set bit at a column c at any line means that instruction represented by that line has a dependency on the instruction at line c. This could be a data dependency, memory dependency or a control dependency. The matrix is divided into four sections. The first section (S-ALU) is reserved for instructions which need a physical destination register such as arithmetic/logic and load instructions and each row of the section corresponds one-to-one to a physical register. The second section (S-MEM) is used

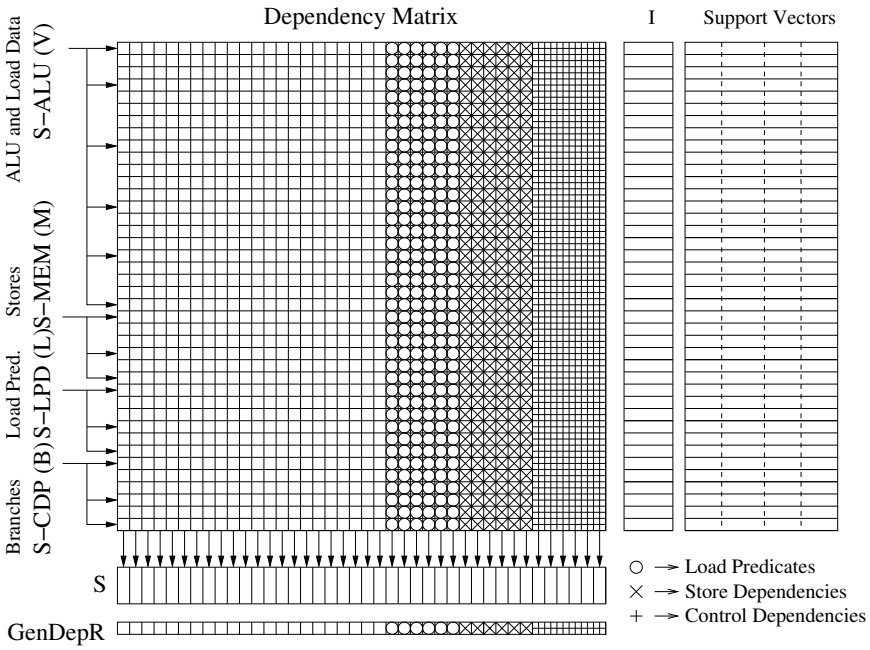


Figure 3.4: Demand Driven Matrix Scheduler[2]

for memory dependencies and provides one row per in-flight store instruction. The third section (S-LPD) is used for load predicates. A load predicate is assigned to each load instruction to represent its memory dependencies. Finally, the fourth section (S-CDP) is used to track control dependencies and provides one row per in-flight branch instruction. S-MEM, S-LPD, S-CDP do not provide physical registers. Assuming the number of physical registers is V, number of store queue entries is S, number of load predicates is L and the number of in-flight branches is given by B, the dependence matrix will have $T = V + S + L + B$ rows and columns, i.e., TxT.

Each line in the matrix represents an instruction, either completed or waiting to be completed. An instruction occupying a matrix line implies a hold on a physical register. A matrix line is only released when the physical register corresponding to

this line is released. The obvious exceptions are load predicate, branch and store sections in the matrix, which do not occupy physical registers. Branch and store lines are released as they are completed and confirmed. Load predicates are released when their partner load executes.

In the matrix, the OR result of a column c is true if another instruction is dependent on the instruction in line c. The result of a horizontal OR of the B entries indicates if the instruction at that line has any unresolved control dependencies. Similarly, the result of a horizontal OR of L or S entries yields if the instruction has any load or store dependencies.

The renaming subsystem follows Alpha 21264 style [24]. A simple vector of RAM holds instructions until they are retired. This vector is accompanied by several other vectors which provide supporting information to track instruction status in the pipeline and processor state, as well information about which instructions are fused. In addition, LaZy Superscalar includes a general dependency register (GenDepR) to identify persistent dependencies. For instance, every instruction will be control dependent on all unresolved branches preceding them.

3.4.2 LaZy Superscalar Pipeline

LaZy Superscalar's pipeline shown in Figure 3.5 follows identical structures to conventional superscalar processors at the front and the rear of the pipeline. The renaming mechanism has been enhanced to rename all instructions. Since dependency checking

is unified, the machine does not incorporate load queues. A store queue is provided for buffering the speculative values from store instructions until they can retire. The Commit phase of the pipeline commits instructions in program order irrespective of their completion status, as each instruction is flagged to belong to the in-order state. Instructions are retired later in an out-of-order manner as they are completed or squashed. In the following sections, we follow the pipeline flow and describe the operation of each stage in relation to registers and other storage that needs to be updated.

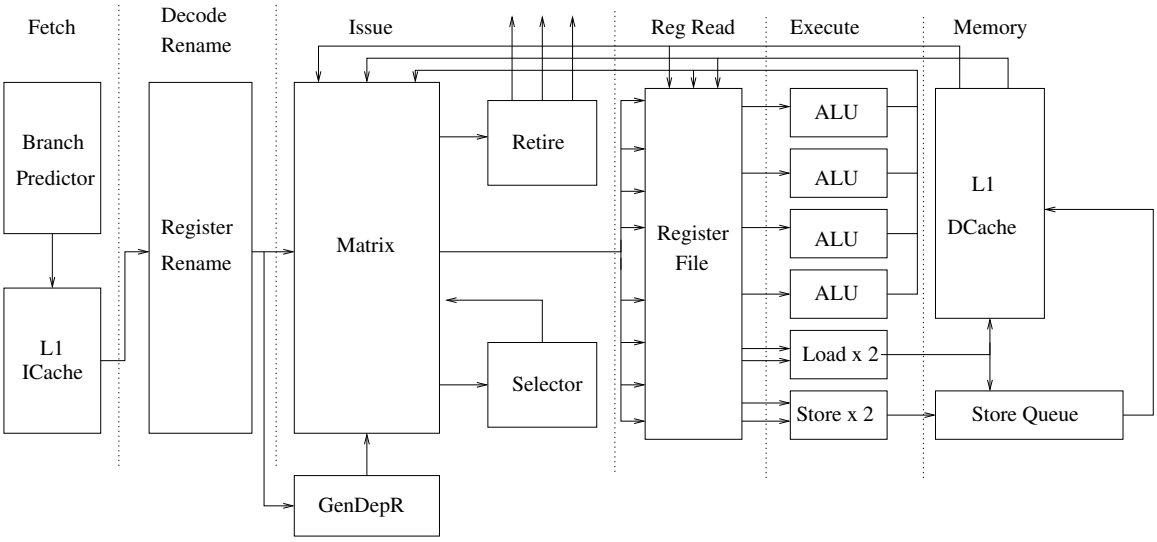


Figure 3.5: LaZy Superscalar Pipeline[2]

Instruction Fetch and Decode The instruction fetch unit supplies the rest of the processor with a predicted instruction stream as in a conventional superscalar processor. Instructions which complete two separate operations, such as memory instructions (address computation and memory access) and jump and link instructions

(set return address and change PC), are dynamically split into two separate instructions for each operation. Memory instructions are dynamically split into an addition and memory operation only if the immediate value in the original memory instruction is nonzero. From then on, the split instructions are treated as separate instructions in the pipeline. This assumption also implies that for multi issue fetch units, each split instruction counts as two instructions fetched. For instance, an 8 issue fetch unit can only fetch 4 store word instructions if their immediate fields are non-zero.

Renaming LaZy Superscalar renames all instructions as described above. Therefore, there will be a stall if there is no free matrix line of the requisite type (value producing, load predicate, store or branch). One special case is a load instruction which requires a data result as well as a memory access result. As a result, load instructions are assigned two registers: one physical register for storing their data and one load predicate to indicate the memory dependencies. Physical source registers are read through the front-end map table. After renaming, the dependency vector to be inserted into the dependency matrix is prepared. Instructions encode all their dependencies using their source operands and global dependency information from the general dependency register.

Instruction fusion is also done during this stage. Each instruction attempts to read the supporting information vectors of its source operands to identify if they are able to fuse to their sources. If fusion availability is detected, appropriate pointers are added to the supporting information vectors for both instructions. In addition,

the dependency matrix is updated to reflect the new dependency relationship formed between the fused instructions. If an instruction A is fused to instruction B, A should no longer be dependent on B, as they will execute in a fused fashion. As replacement, A should be dependent on the operands of B, and B should be dependent on any remaining operands of B. All such changes are reflected in the new dependency vector to be inserted into the matrix and the prepared dependency vector is then passed along to the matrix to be inserted in the slot obtained from the renamer. At the same time, the instruction's matrix line number is inserted into the ROB, the instruction is written into the instruction buffer, and appropriate state control vectors are updated to reflect the instruction's state. The general dependency vector is also updated to reflect any new global dependencies.

Instruction Selection In this stage, instructions are selected for execution based on information output from the DEPMAT. For an instruction to be selected for execution, another instruction must be dependent on that instruction. We call such an instruction a demanded instruction. Being demanded is not the only requirement for an instruction to issue - all its operands must also be ready. If the processor is currently in the process of recovering from an exception, operand readiness is not sought since the instruction results will not be used. If any instruction has a fused pair, the paired instruction is also sent for execution in the same unit.

When more instructions are available for execution than available execution units, instructions without control dependencies are given preference, as they are among the

oldest instructions in the processor and pose no chance of being wasted execution due to a misprediction. Otherwise, instruction selection is done through the physical order of instructions in the matrix.

Execution Instructions read their registers, execute their operation and write back their results in this stage of the pipeline. A completed instruction updates all relevant information in the support vectors to identify completion and updates to the state. In addition, any instructions which are part of the general dependency register are cleared out. Finally, the completed instruction's line in the DEPMAT is cleared since a completed instruction by definition is no longer dependent on any value.

ROB Commit When an instruction reaches the head of the ROB, it leaves immediately after doing some bookkeeping with a few exceptions. Branch instructions must check to see if they've completed before leaving the ROB. A successfully speculated branch instruction will clear its corresponding column in the matrix to indicate the resolution of the control dependency it represents. A mispredicted branch will trigger the misprediction recovery mechanism.

During misprediction recovery, the processor needs to do the following: (1) The retirement map table (RMAP) must be copied to the front end map table (FMAP); (2) Since everything fetched after this branch is incorrect, and branches and stores do not leave the ROB until they are complete, any remaining branch and store instructions in the matrix must be discarded and all corresponding bits must be reset;

(3) Any remaining instructions which depend on the mispredicted branch (encoded by DEPMAT) are marked as an exception; (4) Fetch, decode and rename stages as well as the ROB must be flushed.

Note that the only selective operation we have to do to recover from a misprediction is the modification of some support vector information to indicate an exception. The instruction retire stage will retire these incorrect instructions whenever convenient - no additional logic is required. In fact, with instruction fusion across branches in place, discarding these instructions may require breaking a fused pair, which would be a costly operation in hardware.

Instruction Retire An instruction may be cleared out of the matrix, free its instruction buffer entry and release any registers it's holding when the following conditions are met: (1) Instruction has left the ROB (or the instruction had an exception bit); (2) No other instruction depends on the instruction (indicated by the demand signals on the DEPMAT); (3) The instruction can no longer possibly be part of the in-order state;

Any instruction fitting this criteria is guaranteed to have no more effect on the output of the processor. Therefore, all resources used by these instructions are immediately released and all control bits pertaining to their operation are cleared. Note that an instruction being complete is not a requirement for it to retire.

Deadlock Prevention A demand-driven processor that holds instructions for an arbitrary amount of time may be at a risk of deadlock. Here we will show that LaZy Superscalar will never deadlock given appropriate resources. Stores are demanded automatically in LaZy Superscalar, and on completion will set themselves as ready to retire. While not automatically demanded, branches also set themselves as ready to retire after confirming their speculation result. This release scheme ensures there will be no deadlock due to load predicate, branch and store line unavailability. As long as the processor contains at least one more physical register than double the number of logical registers, there will also be no deadlock for result producing instructions. Consider the pathological case of a series of n load immediate instructions each writing to a different logical register in a machine with n logical registers. None of these instructions would demand another. However, the next result producing instruction has to either demand one of those load immediate instructions, in which case that instruction will get executed and retired, or, the instruction will end up using a logical register already in use. In this case, the previous definition of the register will be marked to be dead and squashed.

Incorrectly applied fusion may cause dependency cycles which will cause deadlocks. An example case is given in Figure 3.6. Instructions i_1 and i_2 are dependent on i_0, but i_1 is not a fusible instruction. Instruction i_2 is additionally dependent on i_1. If i_2 is fused to i_0, the i_0,i_2 pair must now wait for i_1 to execute. However, i_1 also can't execute since it is dependent on i_0. To prevent such dependency cycles,

Inst.	Code Sequence	Fusion Status	Action
i_0	add r1, ...	Fusible (ALU)	Awaiting demand
i_1	lw r2,r1	Not Fusible (Load)	Demands r1
i_2	add ...,r1,r2	Fusible (ALU)	Demands r1 and r2

Figure 3.6: Fusion Dependency Cycle Example[2]

we follow the policy of marking each instruction as non-fusible each time fusibility information is read during instruction rename. With this policy, i_0 would be marked as non-fusible once i_1 reads its sources during renaming. Therefore i_2 will not fuse to i_0 since at this point i_0 is marked to be a non-fusible instruction.

3.5 Evaluation and Experiments

In order to evaluate LaZy Superscalar, we simulated a typical superscalar processor as our baseline as well as LaZy Superscalar itself. Simulators were automatically synthesized from descriptions written in the ADL processor description language [32]. Both simulators are cycle accurate and their ADL implementations respect timing at the RTL level. The baseline processor shown in Figure 3.7 uses centralized scheduling using broadcasting and wake-up/select is completed in a single cycle. Load and store instructions are issued directly to memory units since address computation is done via splitting the computation into another instruction.

We kept the processors as identical as possible. Both processors use identical fetch

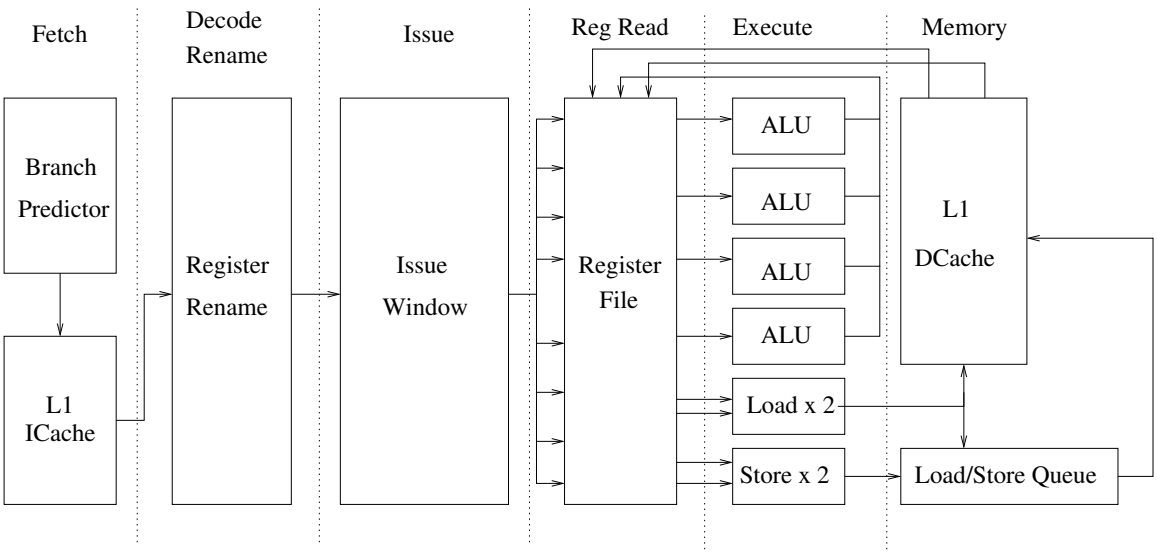


Figure 3.7: Baseline Superscalar Pipeline[2]

engines, fetch, decode and execute the same number of instructions and have identical execution units. It should be noted that both processors also have the same number of read and write ports on their register file. If LaZy Superscalar attempts to execute a fused instruction and there are not enough ports to write both results during that cycle, it will stall. In order to have a fair comparison in terms of issue capability, LaZy Superscalar is provided with a 32 entry store buffer and 32 load predicates as it does not have a load queue and the baseline is given a 64 entry load-store queue. The matrix implementation faithfully implements the operation of the matrix at the bit level.

Experimental parameters are summarized in Tables 3.2 and 3.4. Baseline superscalar enjoys full age based scheduling (oldest ready instruction in the window always schedules first). LaZy Superscalar issues instructions which are dependent on no

branches with priority. The matrix in LaZy Superscalar already provides a single bit output that is 1 if an instruction is dependent on any branch. This issue policy lets LaZy Superscalar approximate age based scheduling. Otherwise, instructions are issued based on their location in the matrix.

Simulation Architecture		
Parameter	**Baseline**	**LaZy**
Front End Width	8 wide	
Commit Width	16 wide	
Issue Width	8 wide	
Issue Window Size	128 entries	N/A
Load Predicates	N/A	32
Execution Units	4 int/fp units	
Memory Units	1 Load/1 Store - 2 Load/2 Store	
Rename Registers	128 registers	
Reg. File Ports	16 read, 8 writes	

Table 3.2: Architectural Parameters Used in Experiments (Part 1)[2]

We executed Spec2006 integer benchmarks which were compiled using gcc version 4.3 with the highest optimization setting (-O3). The software environment used is Binutils version 2.22. Binaries were compiled to MIPS instruction set for Linux kernel 2.6. O/S kernel was not simulated but C library code was included in the simulation. uClibC version 0.9.33 was used to link the benchmarks. We ran the ref inputs for the given benchmarks for 500 million instructions for cache and branch predictor warm

up, then for an additional 1 billion instructions to gather performance and other data.

Simulation Architecture		
Parameter	**Baseline**	**LaZy**
In Flight Branches	16	
Load/Store Queue	64 entries	N/A
Store Queue	N/A	32 entries
Reorder Buffer	176 entries	
L1 Data Cache	32KB 2-way, 1 cycle lat.	
L1 Inst Cache	32KB 2-way, 1 cycle lat.	
L2 Unified Cache	512KB 8-way, 12 cycle lat.	
Main Memory	80 Cycle lat	
PHT Size	16KB	
Branch Prediction	GShare with 4KB BTB	
Mispred. Recovery	4 cycles	

Table 3.4: Architectural Parameters Used in Experiments (Part 2)[2]

Fused instruction distribution over the benchmarks and the performance data are shown in Figure 3.8. The data has been collected by running all input sets for a particular benchmark and taking the average of each run. As can be seen, a large fraction of total instructions are fused successfully using the implemented LaZy scheduling algorithm. However, as expected, fusing a large number of instructions does not necessarily lead to improved performance. Bzip2 is an exception showing a wide range of performance depending on the input set (e.g., chicken.jpg: 19.63%, liberty.jpg: -

10.18 %). Our investigation yielded that the simulation parameters of warming-up for 500,000M and simulating 1B instructions is not a good fit for this benchmark as it cannot finish loading the liberty.jpg in 1.5B instructions, therefore it still is in its initialization phase. Gcc, mcf and perlbench do not show a commensurate increase in performance to that of number of fused instructions. On the other hand, some benchmarks show a larger than expected performance increase, given the number of fused instructions. This is due to the fact that in these benchmarks, the majority of fused instructions are on the critical path. Collapsing such dependencies enables available dependent parallelism to be harvested significantly earlier in the program flow. As LaZy Superscalar is an unguided dependency collapsing technique, we rely on the aggressive fusion technique to direct fusion towards the critical path.

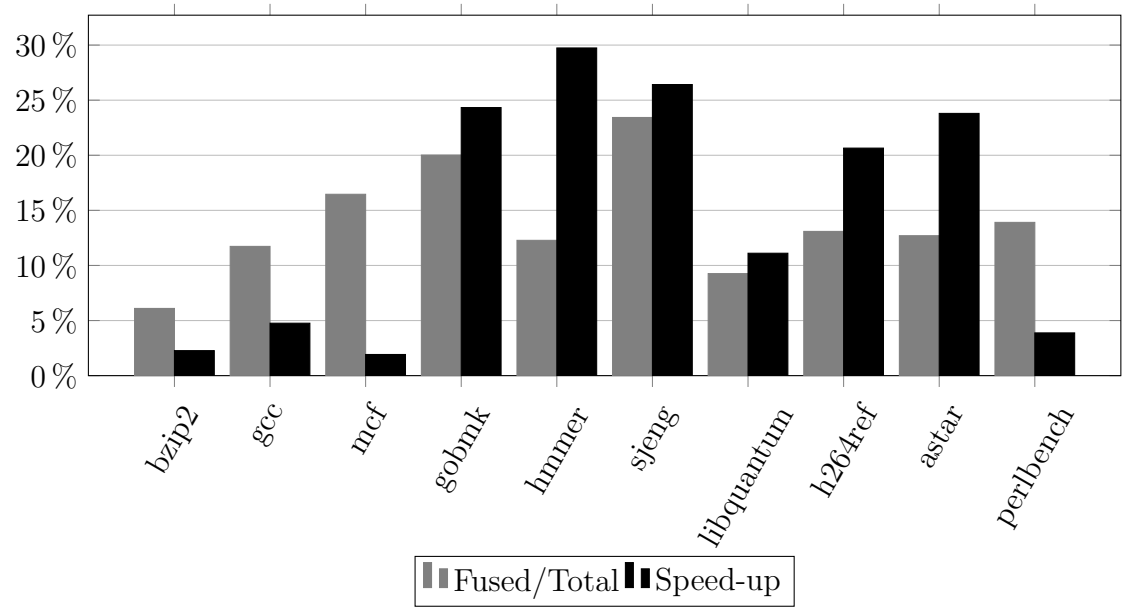


Figure 3.8: Fused Instructions as Fraction of Total and LaZy Superscalar Speed-up

We show an example fragment from the P7Viterbi function in the hmmer bench-

mark in Table 3.5. In the fragment, 3 out of 12 instructions are fused (25%), which yields a 50% speed-up. Fusing shortens a dependence chain of length 4 ($i06 \rightarrow i07 \rightarrow i08 \rightarrow i09$) to 2. Hmmer spends 99% of it's execution in this function, and P7Viterbi contains many similar sequences back to back and in loops.

Excerpt from P7Viterbi in Hmmer		
Instruction	**Baseline Ex Cycle**	**LaZy Ex Cycle**
`i01 lw $2,36($fp)`	1	1
`i02 sll $2,$2,2`	2	2
`i03 lw $3,80($fp)`	1	1
`i04 addu $2,$3,$2`	3	2 (fused to i02)
`i05 lw $4,36($fp)`	2	2
`i06 li $3,1073676288`	1	1
`i07 ori $3,$3,0xffff`	2	1 (fused to i06)
`i08 addu $3,$4,$3`	3	2
`i09 sll $3,$3,2`	4	2 (fused to i08)
`i10 lw $4,92($fp)`	2	2
`i11 addu $3,$4,$3`	5	3
`i12 lw $4,0($3)`	6	4

Table 3.5: Execution Profile Fragment from P7Viterbi in Hmmer[2]

Since LaZy Superscalar only fuses integer instructions, we focus on the integer benchmarks in the Spec2006 set. For completeness, we also evaluated the floating point benchmarks. FP benchmarks get an average performance improvement of 9.06%. Performance profiles are similar to the integer benchmark set.

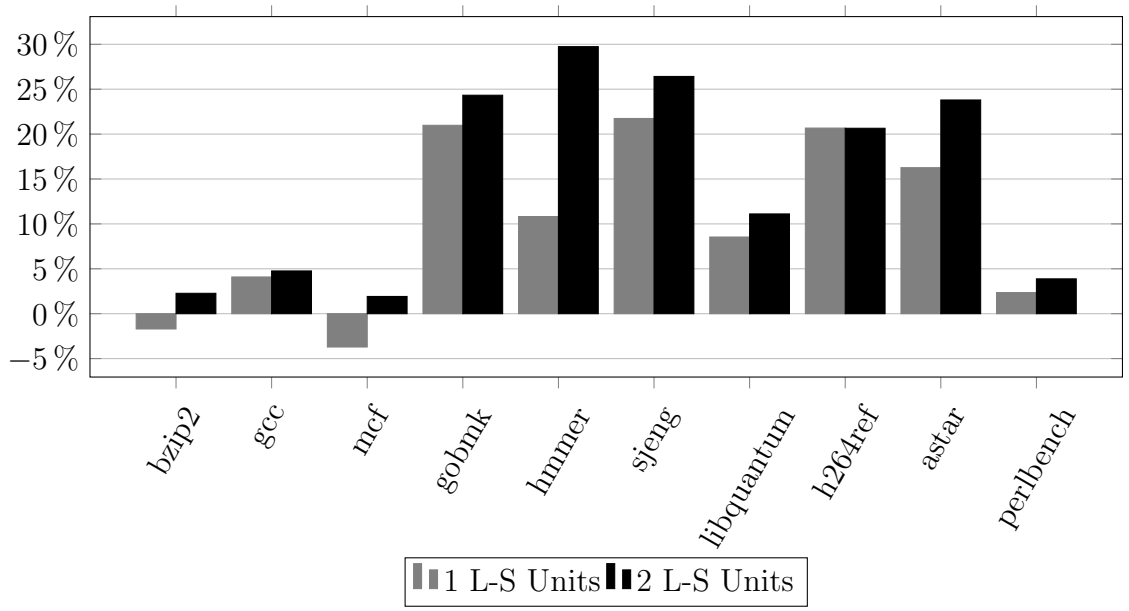


Figure 3.9: LaZy Superscalar Speed-up with Different Load/Store Units

Dependency collapsing techniques improve available parallelism by allowing the processor to harvest *dependent parallelism*. However, sometimes the additional parallelism cannot be harvested due to a lack of resources. In LaZy Superscalar, dependency collapsing comes at the cost of additional delays due to the demand-driven, lazy execution paradigm. If the additional parallelism cannot be harvested, clearly LaZy Superscalar will do poorer in performance. To illustrate the point, we varied the number of load-store units between 1-2. The result is shown in Figure 3.9. With one load and one store unit, which may not be able to harvest the available parallelism in most of these benchmarks, LaZy Superscalar actually loses performance in two of the benchmarks whereas increasing the number of load and store units in both the LaZy Superscalar and the baseline superscalar yields results in which LaZy Superscalar is clearly superior across the entire suite.

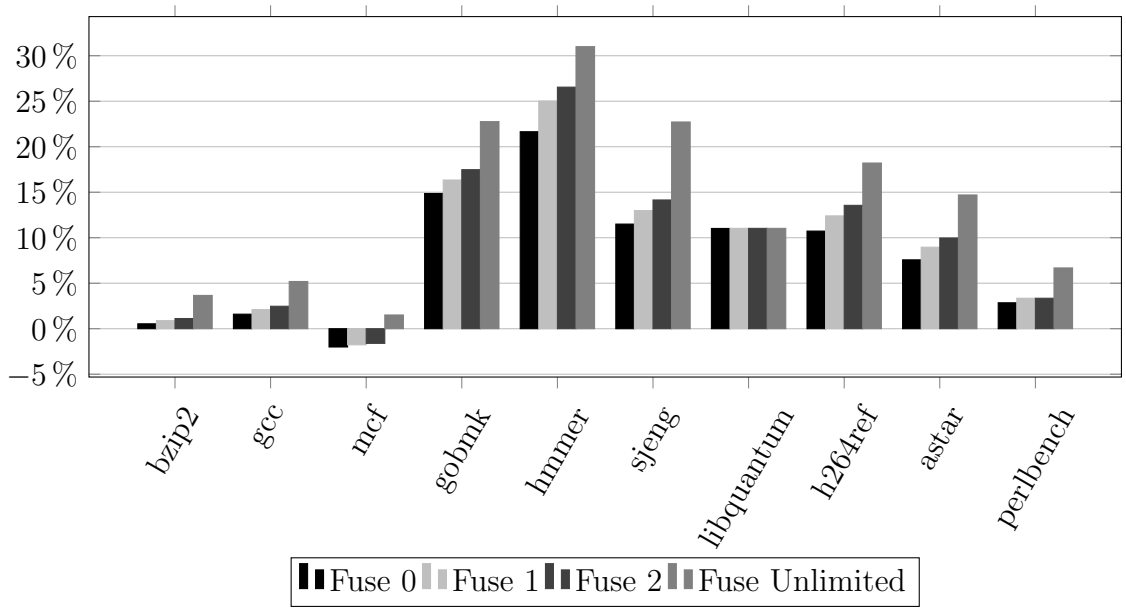


Figure 3.10: LaZy Superscalar Speed-up with Fusing Over N Branches

We also evaluated a limited version of LaZy Superscalar by disabling instruction fusion when there is a branch between the two instructions (Fuse 0), when there are no more than a single branch (Fuse 1), no more than two branches (Fuse 2) and unlimited number of branches (Fuse Unlimited) with the goal of showing the significance of collapsing *distant* and *mutable* dependencies (Figure 3.10). While there are benchmarks which can benefit from fusing in a single fetch block, without exception all benchmarks benefit from fusion across branches, validating our motivation to develop a technique which can collapse *distant* and *mutable* dependencies.

3.5.1 Power Analysis

We incorporated power models and estimated the power consumption for both LaZy Superscalar and the baseline. Power values have been obtained by adapting Wattch[5]

to the ADL simulator framework. The power results have been validated against the McPAT[26] tool tested with a very similar superscalar pipeline to ensure correctness. The breakdown of power consumption is shown in Table 3.6 in watts. LaZy Superscalar consumes less total power in four of the reported benchmarks. In general, LaZy Superscalar consumes more power for ALU operations as expected, but makes up for it through reduced power of the matrix implementation. This information agrees with Safi et al.[43]'s work on the physical characteristics of a matrix scheduler. LaZy Superscalar appears to consume more power when performance increase is high, with mcf and sjeng being exceptions. Libquantum consumes more power in LaZy Superscalar due to increased number of data cache accesses.

The calculated energy delay product (EDP) of LaZy Superscalar over the baseline implementation is 0.92 on average. Assuming both machines can be implemented at the same clock speeds, we believe LaZy Superscalar promises to be a better approach than conventional eager scheduling with its lower EDP.

	Baseline				LaZy Superscalar			
	Total	**Inst Win**	**ALU**	**RegFile**	**Total**	**Inst Win**	**ALU**	**RegFile**
bzip2	24.3419	5.3863	1.6016	1.846	23.7724	2.95	2.6083	0.8778
gcc	26.739	6.1104	2.0164	2.0082	24.7298	2.85	2.8366	0.8912
mcf	18.0578	3.7762	1.1973	1.2332	18.2716	3.012	1.7223	0.5534
gobmk	22.6356	4.8681	1.59	1.7178	24.5608	3.01	2.836	0.8543
hmmer	26.3296	5.6902	2.0607	2.2308	28.359	2.89	3.5401	1.1101
sjeng	35.6064	7.4474	2.5745	2.4457	34.1901	2.845	4.0962	1.1526
libquantum	35.6022	7.8922	2.5969	2.8325	43.0629	3.01	5.4218	1.9866
h264ref	27.0862	6.0292	2.0526	2.2135	28.2634	2.802	3.4117	1.1523
astar	30.517	6.9825	2.1313	2.5826	33.5478	2.92	3.9513	1.3287
perlbench	22.656	5.4913	1.825	1.7021	20.8651	2.665	2.4505	0.7612

Table 3.6: Power Analysis (watts)[2]

Chapter 4: Collapsing Resource Dependencies

In this chapter, we discuss collapsing dependencies which arise from architectural constructs in the micro-processor as opposed to dependencies which arise from program semantics[1]. The effects of such dependencies are most pronounced when the dependent data is stored in physically distinct locations. Further, such architectural dependencies also play a significant role in scheduling decisions. In this chapter, we explore a novel scheduling technique to allow multiple physical structures for architectural data storage in a superscalar processor with minimal impact on performance. This technique allows the issue window, register files, execution units, bypass networks and potentially memory units to be clustered without requiring explicit inter-cluster communication.

[1]Parts of the material contained in this chapter will be submitted for publication in a conference.

4.1 Overview

As superscalar processors grow in the number of available register read and write ports, cache size, issue width, number of available execution units and size of bypass networks, the power consumption and access latency of these structures grow in a larger-than-linear fashion. This physical reality prevents computer architects from designing larger monolithic processors, even when there is additional parallelism available to be extracted with larger structures. Since these structures poorly scale when grown, but linearly scale when multiple instances of them exist (such as having multiple register files), some architects have chosen to duplicate smaller instances of these structures, effectively dividing the processor into *clusters*.

Such clusters allow scaling resources that would be too prohibitive to scale in a monolithic fashion. On the other hand, clusters require additional policies and rules in place to allow inter-cluster communication. For instance, if a processor contains two separate register files instead of a single one, operands for an instruction must be potentially collected from multiple register files. Clustering therefore introduces additional dependencies to the processor that are not defined by the program semantics.

Historically, additional dependencies introduced by clustering are handled in a number of different ways. One common approach is to allow for extra delay in accessing multiple structures such that data from a distant cluster can be appended

to existing data. Other techniques attempt to minimize this additional latency by dynamically scheduling dependent instructions to the same cluster when possible.

In the following section, we introduce a novel approach where an instruction is scheduled to a cluster such that instructions producing its source operands *have not* executed in that cluster. In other words, we only permit each instruction to write to a different cluster than the one in which it executes. We show that doing so allows for an arbitrary number of clusters of register files, bypass networks and issue windows to be used in which all inter-cluster data dependencies are transparently handled during the rename stage. Hence, each cluster acts as a truly independent computation unit within the same processor.

4.2 Introduction

In a superscalar processor, increasing the issue width requires scaling up many structures, such as the number of register read and write ports, the size of bypass networks, as well as the number of cache ports. All of these structures yield a non-linear increase in the power consumption and access latency when scaled-up [56], which can be remedied by clustering these structures.

To the best of our knowledge, all existing clustering techniques are based on the premise of gathering the data values from another cluster when the required data is not available in the cluster in which the instruction is scheduled to execute. However, they differ on their scheduling techniques to minimize such communication.

Gathering of data from other clusters is accomplished by providing extra connectivity in the form of inter-cluster bypass networks as shown in Figure 4.1. In addition to this inter-cluster bypass network, each cluster contains a local bypass network attached to its own functional units, as shown in the figure. The local bypass network provides operands of dependent instructions executing in the same cluster whereas register values not found in the cluster are transferred from other cluster(s) using the inter-cluster bypass network. Transferring values over the inter-cluster bypass network typically introduces additional penalties.

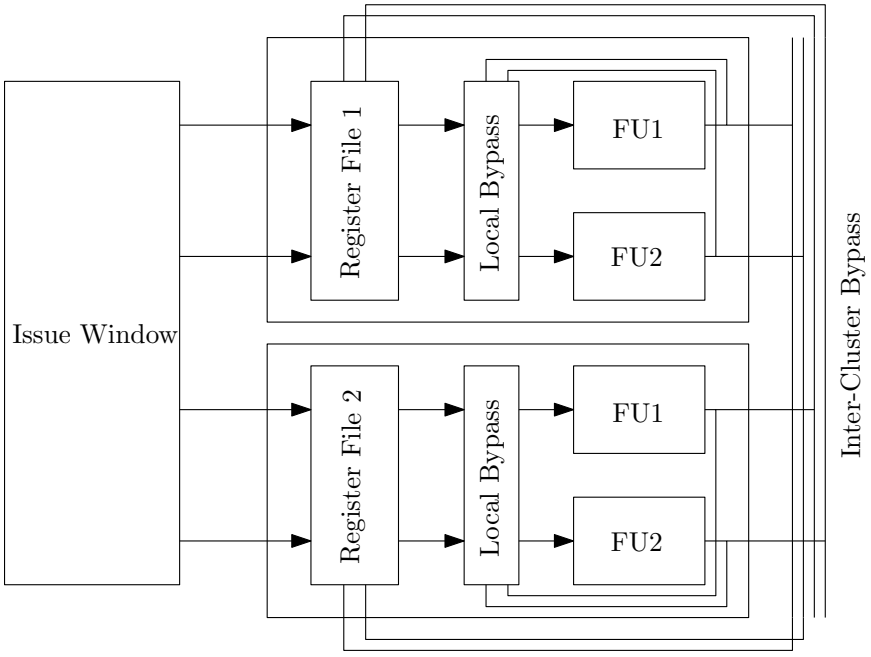


Figure 4.1: Typical Clustered Register File Architecture

In this chapter, we present a novel approach where the local bypass network is eliminated and a uni-directional inter-cluster network instead is utilized to connect the clusters together as shown in Figure 4.2. We call this organization a *uni-directional*

cluster.

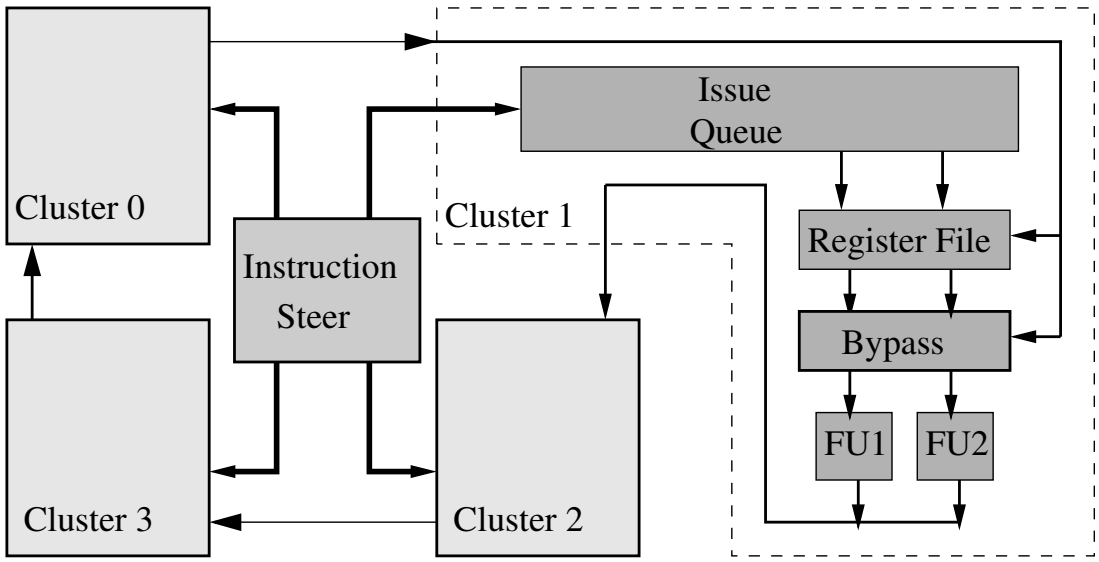


Figure 4.2: A 4-cluster Uni-directional Cluster Architecture

In this approach, a given cluster always writes to other cluster(s), and two dependent instructions are never scheduled to execute in the same cluster. Since there are no dependent instructions in a given cluster, there is no need to have a local by-pass network, as the operands of a given instruction will either be locally read from the register file, or, they will come from another cluster. Hence, the by-pass network of a given cluster is connected to the output of another cluster in which producers of the instructions in this cluster execute. In other words, in our clustered architecture, the physical layout of the architecture represents a dependent chain of operations and the instructions are scheduled onto each cluster based on their dependencies. As a result, the data-flow among instructions is naturally mapped to the available architectural communication paths.

The steering of instructions into the appropriate clusters is easily accomplished by modifying the renamer which keeps track of which clusters have the desired operands to minimize communication. If a desired operand is not available in the target cluster where an instruction will be executed, a copy operation is scheduled to the preceding cluster. Since the communication is uni-directional and computed data values are not locally consumed, global and local traffic do not compete. The 4-cluster uni-directional cluster architecture shown in Figure 4.2 utilizes four register files, each with two read ports and one write port. This is an enormous amount of simplification compared to a 4-wide superscalar which would need a register file with 8 read ports and 4 write ports, provided the ILP can be sustained under the new paradigm.

As it can be seen, the clustering mechanism shown in Figure 4.2 is extremely efficient for handling dependence chains. Consider a dependent chain of instructions *i, j,* and *k* such that *j* is dependent on *i* and *k* is dependent on *j*. If *i* is scheduled to cluster zero, *j* is scheduled to cluster one, and *k* is to cluster two. There are no delays and full utilization of clusters is accomplished. This design however suffers when we have a dependence sequence such that *i* produces a value, which is consumed by both *j* and *k*. In this case, instruction *i* can be scheduled to cluster zero, and *j* to cluster one. However, scheduling *k* also to cluster one increases pressure on this cluster when cluster two would be idle. Scheduling *k* to any other cluster requires a copy operation to be scheduled to propagate the value from cluster one. In order to remedy this situation, we can permit a cluster to write to more than one cluster. This approach

in effect duplicates the produced value in multiple clusters, hence both j and k can simultaneously proceed, each executing at the target clusters of cluster zero. Under this organization, due to multiple destinations from each cluster, each cluster needs a register file with two read and two write ports and each by-pass network needs to have two access ports.

In this clustering mechanism, it is possible to cluster only the register file and the execution units, the register file, execution units as well as the issue queues, or, even the entire back-end of the processor, including the reorder buffer. As it can be seen, the design space for this clustering mechanism is quite large with each design point having different advantages and disadvantages. Before we explore various designs in that space, we first discuss the rename and steer mechanism in the next section.

4.3 Instruction Steering

Given that an instruction executing in a given cluster can only read its source operands from the register file in that cluster, or, from the by-pass connected to the upstream cluster, a given value needs to reside in more than one register file, and we need to keep the values coherent. We therefore partition the physical register name space, allocating an equal chunk of physical names to each cluster. A very easy way to partition the name space is to use physical register number modulo number of clusters to identify which cluster it belongs to. For instance, in a two-cluster organization, physical registers with an even number would belong to cluster zero, and physical

registers with an odd identifier would belong to cluster one.

Once this partitioning is done, we can map a given logical register to multiple physical registers, each residing in a different cluster. Since each logical register can map to physical registers up to the number of clusters (i.e. in a four cluster organization, a logical register can map to up to four registers, each being in a different cluster), we extend the register map table to have as many columns as there are clusters. In this manner, the renamer is guaranteed to find at least one mapping for a given logical source register, but it may find mappings up to the number of clusters in the organization.

With this set-up, instructions can be easily steered to a cluster at the rename stage. When an instruction is processed, the renamer checks the extended map table to get the current logical to physical mapping. For each operand, this read may return multiple physical registers, from which we can precisely know in which cluster(s) a given source operand is available. Given this information, we steer the instructions as follows:

1. If the current instruction is a single operand instruction, it is dispatched to the cluster with the smallest utilization which has its operand. The utilization for each cluster is easily checked from the size of the available register pool for that partition of the register name space.

2. For dyadic instructions, if there is a cluster which has both of the operands, the instruction is dispatched to that cluster. Should multiple clusters satisfy

this condition, the instruction will be scheduled to the cluster with the lowest utilization.

3. If there is no cluster which contains both of the operands of a dyadic instruction, a cluster which contains one of the instruction's operands is selected, and a *copy* instruction is injected into the pipeline. The copy instruction is placed to execute in the upstream cluster of the selected cluster. For this purpose, the cost in terms of number of copy operations needed for the other operand is calculated for each of the clusters which has the operand and the instruction is steered to that cluster. As before, the tie breaker is the utilization.

Once a cluster is selected in which the instruction will be executing which we refer to as the *source cluster*, its destination register is allocated from the register pool of the cluster to which it will write, i.e., its *target cluster*.

In this architecture, a *copy* instruction is an instruction that reads from the same logical register that it is writing to, e.g., $\$r = \$r;$, where $\$r$ represents the logical register whose data is being copied to another cluster. Hence, $\$r$ will be copied from the cluster it exists on to the following cluster in the ring, since data propagation is always uni-directional in the architecture. As it should be clear, a copy instruction is equivalent to a no-op. The only special handling required by a copy instruction is that the source physical mapping is not invalidated despite being "overwritten". This change ensures the data is copied, instead of simply being moved. In a two-cluster ring, generating a single copy instruction is always sufficient. In a cluster ring made

up of more than two clusters, multiple copy operations may need to be generated.

During renaming, the copy instruction is emitted alongside the original instruction to be dispatched and there is no particular order that needs to be imposed in issuing them. This is because, once renamed, the consumer instruction will wait for the availability of the data and will be woken up by the select logic once the data becomes available (i.e., the copy instruction executes).

4.4 Renaming of Instructions

Clustering in this manner primarily affects the mechanisms used for renaming and retiring instructions in a superscalar processor. This is due to the fact that a given logical register will have multiple physical registers assigned to it and we need to have the proper mechanisms for the allocation and freeing of these registers. In this section, we describe the details of the renaming mechanism by following a simple example of renaming in a two-cluster uni-directional ring architecture as shown in Figure 4.3 and then illustrate the primary changes which need to be done to the processor pipeline for the retire logic for correct execution.

In our discussion of the algorithm, we assume the register name space is partitioned into two partitions such that even numbered registers reside in cluster zero, and, the odd numbered registers reside in cluster one. Furthermore, the architecture maintains two free physical register lists, one for each cluster. The map table, which is part of the steering logic is outside the clusters and is centrally maintained.

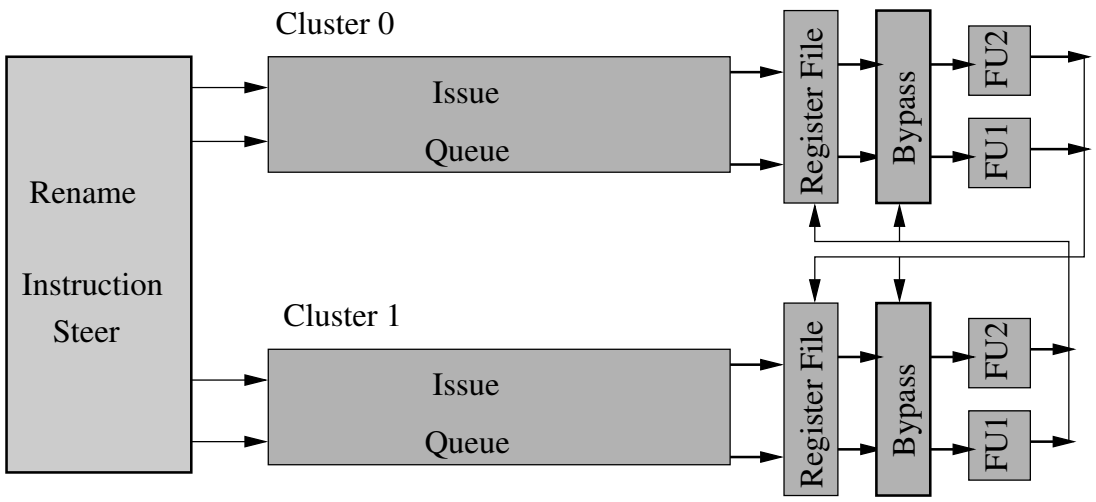


Figure 4.3: A 2-cluster Uni-directional Ring Architecture

Let $r0$, $r1$, $r2$ and $r3$ represent the logical registers used in a program. Further, let $P0$, $P1$, $P2$ and $P3$ represent physical registers assigned to the corresponding register values at the start of the program.

The map table at the start of the program is shown in Figure 4.4. A blank at a given entry of the table indicates that a physical register has not been assigned to that logical register in that cluster (i.e., that column).

Consider the following instruction:

$$i_1 : \$r0 = \$r1 + \$r3$$

A cluster for i_1's execution will be selected as i_1 is being renamed. The physical source registers for i_1 are $P1$ and $P3$, both of which are in cluster C_1. The execution cluster is then selected as C_1. Due to the uni-directional data-flow, the destination register will be assigned from C_0. Therefore, the instruction is renamed as follows:

$$i_1R : \$P4 = \$P1 + \$P3$$

Map Table		
	Cluster 0 (C_0)	Cluster 1 (C_1)
$r0$	$P0$	
$r1$		$P1$
$r2$	$P2$	
$r3$		$P3$

Figure 4.4: Initial Map Table

and the map table is updated as shown in Figure 4.5.

Map Table		
	Cluster 0 (C_0)	Cluster 1 (C_1)
$r0$	$P4$	
$r1$		$P1$
$r2$	$P2$	
$r3$		$P3$

Figure 4.5: Map Table After i_1

Let us consider the following instruction next:

$$i_2 : \$r1 = \$r0 + \$r3$$

i_2's sources are shown to be $P4$ and $P3$, which are in different clusters. In this case, the processor chooses C_0 as the execution cluster. $P3$ then must be moved into

C_0 before execution. The following move instruction is emitted to accomplish this:

$$i_{2m} : \$r3 = \$r3$$

The new move instruction will execute in C_1, since its only operand is in C_1. i_{2m}'s result register will be chosen from C_0, and i_2's result register will be chosen from C_1 due to the uni-directional data-flow. The move instruction combined with the original i_2 instruction are then renamed like so:

$$i_{2m}R : \$P6 = \$P3 i_2R : \$P7 = \$P4 + \$P6$$

The rename operation produces the map table shown in Figure 4.6.

Map Table		
	Cluster 0 (C_0)	Cluster 1 (C_1)
$r0$	$P4$	
$r1$		$P1$
$r2$		$P7$
$r3$	$P6$	$P3$

Figure 4.6: Map Table After i_2

Of note in Figure 4.6, the move instruction does not invalidate the physical register in the front-end map table for $r3$ since each move instruction is guaranteed to not change the value of the register between the clusters. However, as $r2$ is redefined by i_2, all previous mappings at each cluster become invalid.

Retiring Instructions

There are several ways the state maintenance is done in a superscalar processor, such as using a reorder buffer coupled with front and back-end (in-order) map tables, as well as checkpointing. Here, we assume that the processor maintains a front-end map table which is speculatively updated and a retirement map table which represents the in-order program state, which is updated by the retire logic. In such a processor without clustering, each instruction must update the in-order-map table to indicate which physical register contains the most up-to-date in-order version of any of its destination registers. The previous physical register must then be released.

However, in a clustered architecture, each logical register may have valid physical mappings in each cluster. Coherent operation in the presence of multiple copies of a given register is easily accomplished by treating all copies as if they represent a single assignment. When retiring an instruction, we now free all previous mappings of a register regardless of the cluster its defined in, a new definition of a logical register invalidates all existing copies. Accordingly, any freed registers are returned to their own pool in their cluster.

Generated copy instructions need a special treatment as they should not free any registers. Any destination registers allocated to the copy instructions are naturally released when that logical register is redefined.

4.5 Exploring the Design Space

The design space of the proposed uni-directional clustering mechanism is huge. There are many variations which lead to different performance characteristics with a possibly significant variation and impact on the complexity of an actual implementation for each approach.

In this dissertation, we explore the clustering of the register file only, clustering of the register file, the instruction shelves and wake-up/select logic, and finally, the connectivity of the clusters as an independent parameter for each of these designs.

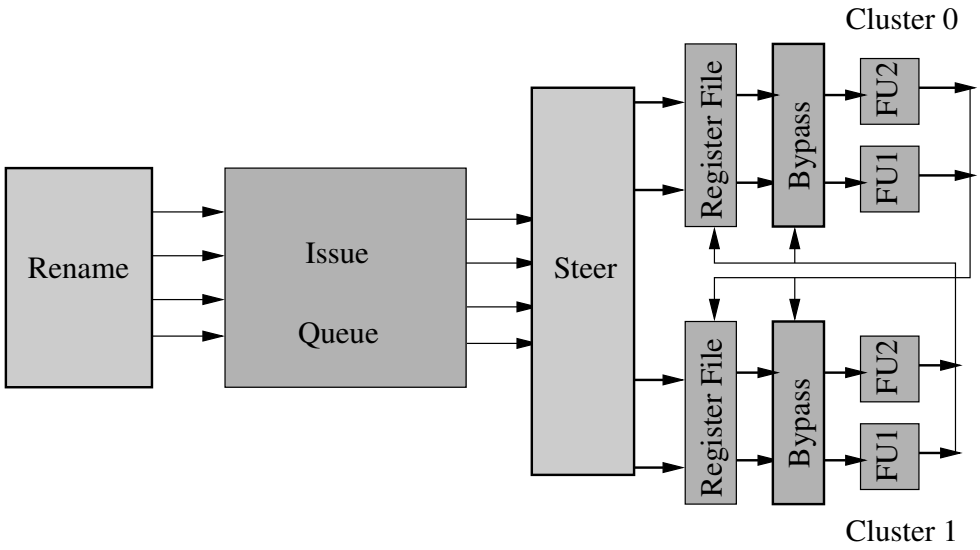


Figure 4.7: Unified window uni-directional two cluster architecture

In its simplest form, the approach can be used to cluster only the register file and the execution units attached to register file lanes. Such an architecture will have a unified instruction issue queue, but have separate register files and execution units

as shown in Figure 4.7. In such a design, the instructions can be steered by using either their source or destination register numbers. The advantage in this design would be better utilization of instruction shelving space (i.e., reservation stations), but the complexity of wake-up/select logic is not reduced, despite the register file being clustered. Combined with an instruction replay mechanism and speculative wake-up, the complexity of the instruction shelves and wake-up/select logic will be similar to that of a unified design.

An alternative organization is to cluster both the register file and the issue window as shown in Figure 4.3. Such a design significantly reduces the complexity of wake-up/select logic at the risk of possibly imbalanced utilization of instruction shelving space. This is a well-known disadvantage of distributed reservation stations. As stated at the beginning, in the rest of the chapter, we explore both designs, but add a third dimension, namely, connectivity.

In the chapter, so far, we have only discussed uni-directional clustering where each cluster writes to the next cluster in sequence. Yet a third dimension is the number of down-stream clusters, i.e., how many clusters should receive a value produced in a given cluster. At the expense of increasing the write ports, a cluster can write to more than one cluster, which may significantly reduce the number of copy instructions that might be needed. As we illustrate later, this aspect is crucial for having a clustered architecture with near zero impact on the total instructions per cycle (IPC) compared to a unified design.

In order to understand the impact of various decisions, we have developed a number of cycle-accurate simulators, which faithfully implement a typical superscalar processor that is clustered by following the primary design space layouts.

Simulation Architecture	
Parameter	**Baseline**
Front End Width	4 wide
Commit Width	8 wide
Issue Width	4 wide
Issue Window Size	64 entries
Execution Units	4 int units, 2 fp units, 2 address computation units
Memory Units	2 Load/2 Store
Rename Registers	320 registers
Load/Store Queue	64 entries
Reorder Buffer	256 entries
L1 Data Cache	32KB 2-way, 1 cycle lat.
L1 Inst Cache	32KB 2-way, 1 cycle lat.
L2 Unified Cache	512KB 8-way, 10 cycle lat.
Main Memory	100 Cycle lat
PHT Size	16KB
Branch Prediction	GShare with 4KB BTB

Table 4.2: Architectural Parameters Used in Experiments

Table 4.2 shows the common architectural parameters used in all experiments,

unless the specific experiment indicates otherwise. In most designs, we study a two cluster variant and a four cluster variant. Assuming only the register file is clustered, compared to the baseline depicted in Table 4.2 which has a unified register file of eight read ports and four write ports, a two clustered architecture (2-cluster) would have two register files with four read ports and two write ports. Similarly, a four-cluster architecture (4-cluster) would have four register files, each with two read and one write ports. Unless stated otherwise, we keep the total number of registers the same in all designs. In other words, our 4-cluster architecture has four register files, each having a total of 80 registers, yielding the same total as the baseline.

We simulated all designs by using MIPS-I ISA without delayed branching. This ISA is very similar to PISA ISA, used by SimpleScalar [6]. GCC 4.9.2 tailored to this ISA is used to compile the benchmarks and generate binary code with the highest optimization ("-O3") set. We choose Spec 2006 as our benchmark suite. All simulation models were designed with Architecture Description Language (ADL) [33]. The ADL compiler can automatically generate the assembler, the disassembler and a cycle-accurate simulator which respects timing at the register transfer level from the description of the microarchitecture and its ISA specified in ADL language.

In order to efficiently simulate our mechanisms, we incorporated Simpoint 3.2 [38, 47] to minimize the simulation time. For each benchmark, a set of checkpoint images were generated where each checkpoint image contains the complete memory data segments, the register file and the program counter (PC). Other architecture related

structures were not included, such as cache, branch predictor, memory dependence predictor, etc. Hence, the simulation of each interval has a cold start. In order to compensate for this effect, we selected a large size, 100 million retired instructions, to simulate each interval. Since each interval simulation was independent of others, we simultaneously simulated all of the intervals to further shorten the simulation time.

4.6 Clustering the Register File Only

Figure 4.8 shows the performance difference between the baseline configuration compared to a configuration where the register file is clustered into two clusters as well the configuration where the register file is clustered into four clusters. In both designs, a unified issue window drives the rest of the processor made-up of split register files and execution units, as shown in Figure 4.7.

As it can be seen, for a 2-cluster, IPC loss is relatively small, yielding a geometric mean of less than four percent, whereas for a 4-cluster, IPC loss is significant, particularly for some benchmarks such as 464.h264ref (20%). This performance loss is directly correlated in most benchmarks with the number of copy operations generated, some of which have excessive number of copies, particularly for 4-cluster configuration. However, this is not always the case. What we observed is, when the ILP in a given benchmark is high, clusters run out of registers, owing to much smaller number of registers per cluster. Furthermore, while in a 2-cluster architecture, both register files eventually start to have a lot of common values, in a 4-cluster, individual cluster

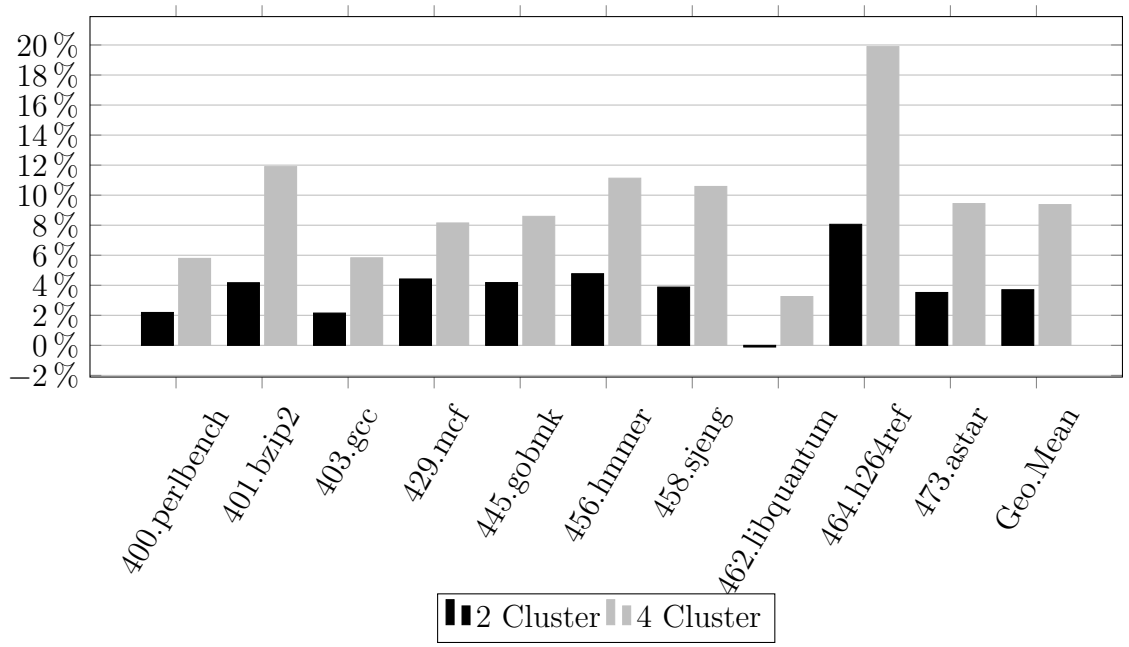


Figure 4.8: IPC Loss with Multiple Clusters

pressure forces instructions to other clusters which will need the data values propagated using copy instructions. Figure 4.9 illustrates the point. As the figure shows, the ratio of copy instructions generated for 4-cluster is significantly higher than a 2-cluster configuration.

4.7 Dual Write Clustering

Reducing the number of copy operations is possible by connecting each cluster to more than one cluster and allowing instructions to write to two physical registers, each in a separate cluster. In this manner, generated values can rapidly propagate to other clusters and the approach reduces the pressure on clusters which have these values. For this approach, the processor pipeline, primarily the renamer, by-pass

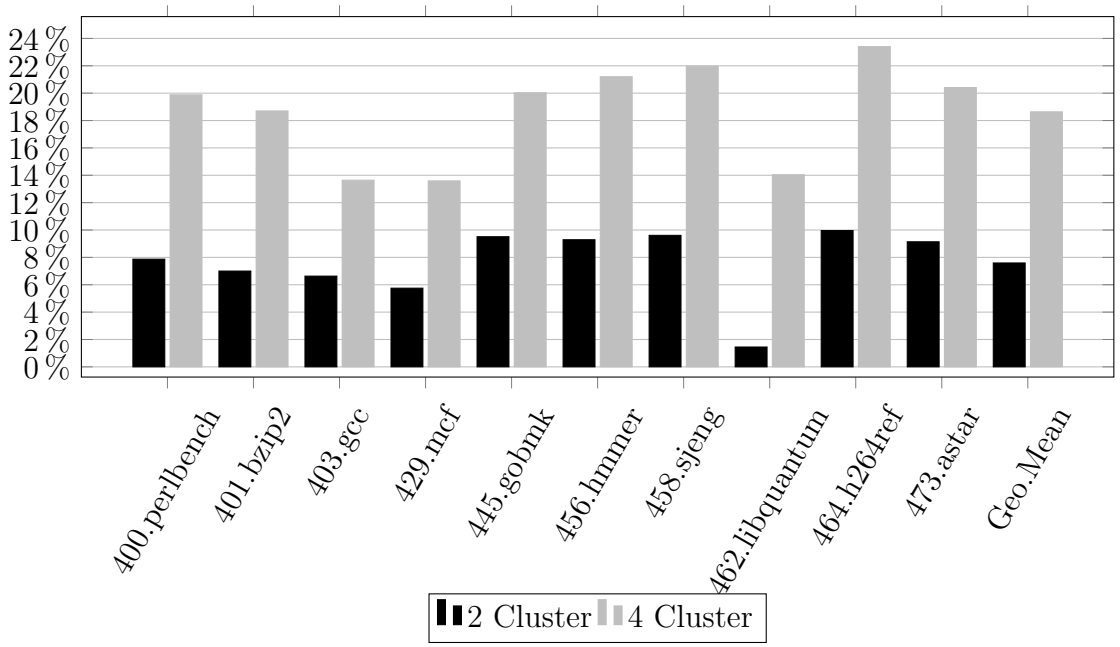


Figure 4.9: Generated Copy Instructions as a Percentage of Total

ports and retirement and register free logic need to be modified to allocate, update, and free more than one register. While the spectrum of the issue of connectivity range from one to the number of clusters, in this dissertation, we evaluate several designs by allowing each cluster to write to two destination clusters. We refer to these configurations as *dual-write*.

Figure 4.10 shows IPC loss compared to the baseline with a dual-write, unified issue window architecture with clustered register files and execution units.

In a dual-write clustered architecture, arithmetic operations and load instructions are allocated two physical registers, one in each target cluster by the renamer. The map table does not need to change compared to our previous design, as a given logical register may be associated with a number of physical registers, namely, as many as

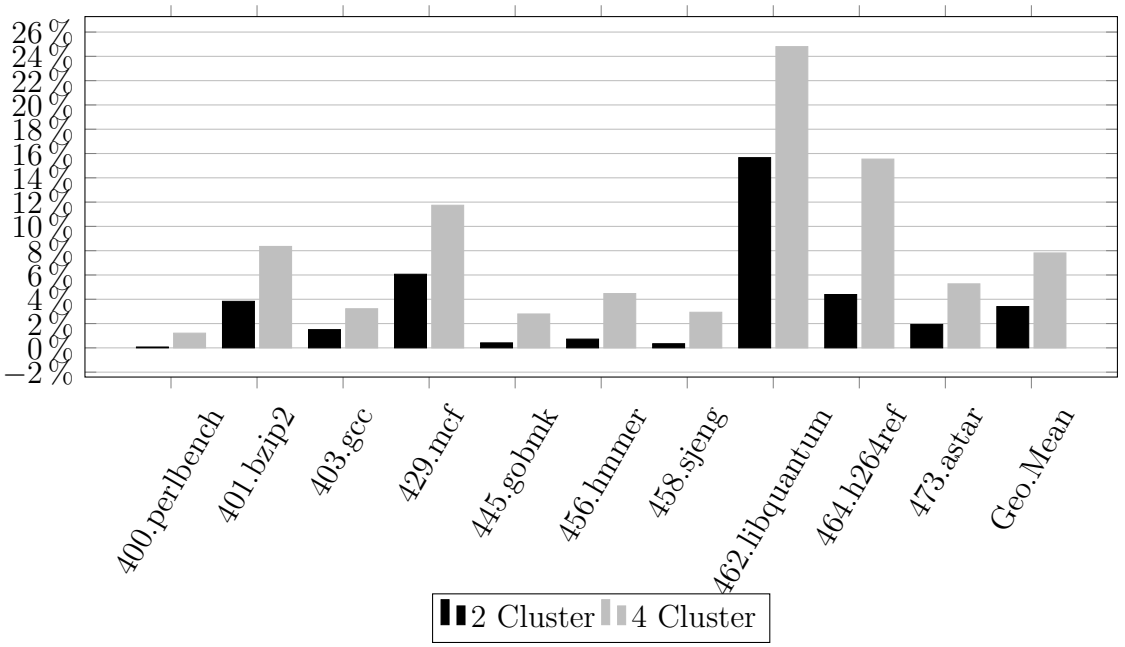


Figure 4.10: IPC Loss with Dual Write

the number of clusters. Accordingly, the by-pass now needs to accept data from two sources. Recall that in a single-write cluster organization, the in-order map table was updated with the destination physical register and all registers in the previous mapping were released. In this configuration, the in-order map table is updated with two destination registers and all registers in the previous mapping are released.

These experiments show that dual-write is effective for most benchmarks yielding a geometric mean of less than 4% for a 2-cluster and less than 8% for a 4-cluster architecture. On the other hand, libquantum shows significant performance loss in both configurations. Our investigation indicates that this benchmark needs a lot of renaming registers. With dual-write, the number of registers in each partition is half (or quarter) of the baseline, leading to many stalls by the renamer for lack of rename

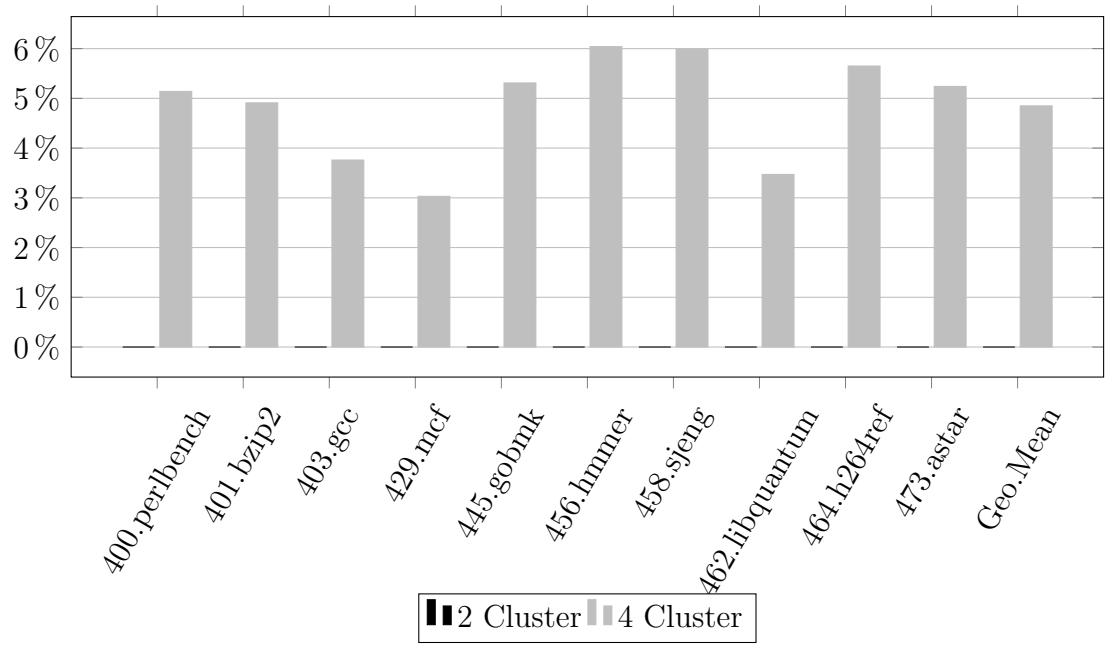


Figure 4.11: Generated Copy Instructions as a Percentage of Total For Dual-Write

registers.

Figure 4.11, which shows the ratio of copy instructions generated for the dual-write experiment. We observe a significant reduction in the number of copy operations across the suite. Compared to the single-write configuration which has 18% copy instructions introduced, dual-write generates less than 5% copy operations.

Further analysis of both the IPC data including key processor stalls, particularly for benchmarks 462.libquantum and 464.h264.ref indicates that many dual-written registers are not read but take-up resources, whereas a large number of them are useful in reducing the required number of explicit copy operations as well as better balancing instruction steering, as instructions can find their operands in multiple clusters. Can we have the advantages of dual-write without the increased register

pressure?

4.8 Dual-write with Lazy Register Allocation

The key concept we have developed in this regard is to allocate two destination registers for each arithmetic and load instruction, but mark them as *primary* and *secondary*. As in a single-write architecture, the primary physical register is removed from the free register pool and it is committed to the current instruction's logical destination register. On the other hand, although the secondary physical register is removed from the free register pool and is assigned to the instruction, it is entered to a spare free register pool. The instruction's reorder buffer entry number is also entered to the spare free register pool as a coupled value with the physical register number. If a subsequent instruction reads this physical register, the register is removed from the spare free register pool and committed to the instruction's logical destination. In this case, it too becomes a primary register. If an instruction's secondary register is *stolen* in this manner from an instruction, the instruction needs to be informed before the register write takes place. The application of this technique means the second register for the dual-write is *lazily* allocated, since the allocation is not finalized until it becomes certain the register will be used.

For this purpose, we add a one bit *secondary register writable* field to each reorder buffer entry. This bit indicates if the corresponding register is still owned by the instruction. When the renamer runs out of physical registers in the main pool

and allocates a register from the spare pool, the writable bit of the owner instruction is cleared. Note that the, secondary register writable field can be coupled with *instruction complete flag* in the reorder buffer for efficient access.

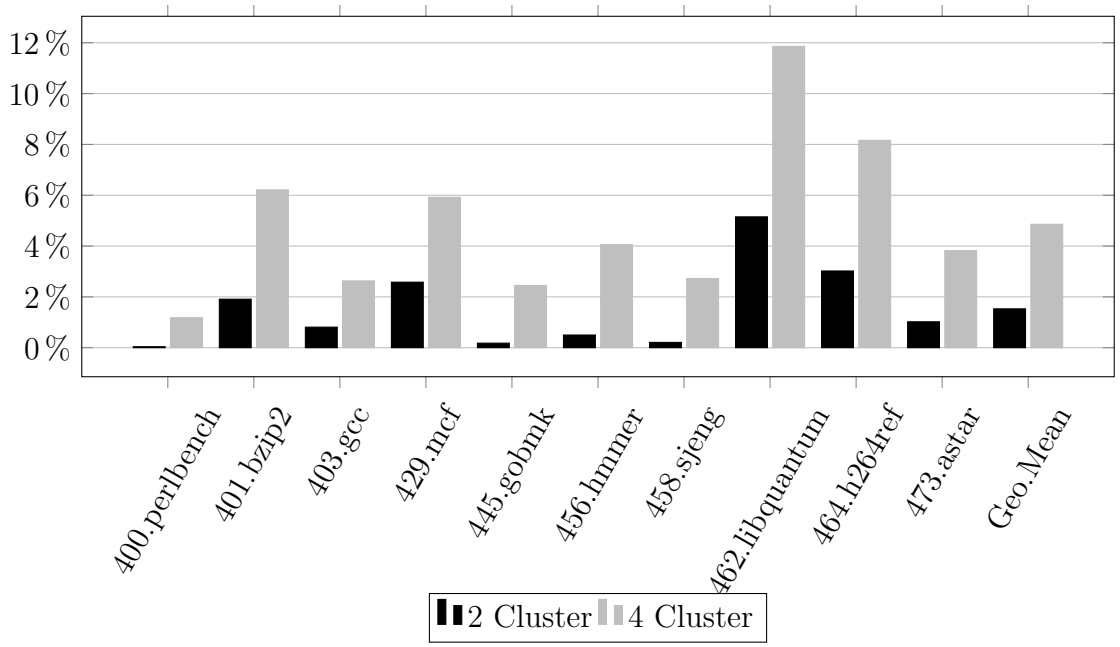


Figure 4.12: IPC Loss with Unified Window - Dual Write, Lazy Allocation

With these changes added, we again collect performance data in terms of IPC. Figure 4.12 illustrates the performance of dual-write with lazy register allocation and Figure 4.13 gives the distribution of copy operations generated under the scheme. These figures show the effectiveness of the technique. Even with a 4-cluster architecture, which means, instead of an 8-read port 4 write port register file, four 2 read port one write port registers files are used, the geometric mean of IPC loss is less than 5%. For a 2 cluster, it is below 2%.

These experiments illustrate that although the geometric mean IPC loss is quite

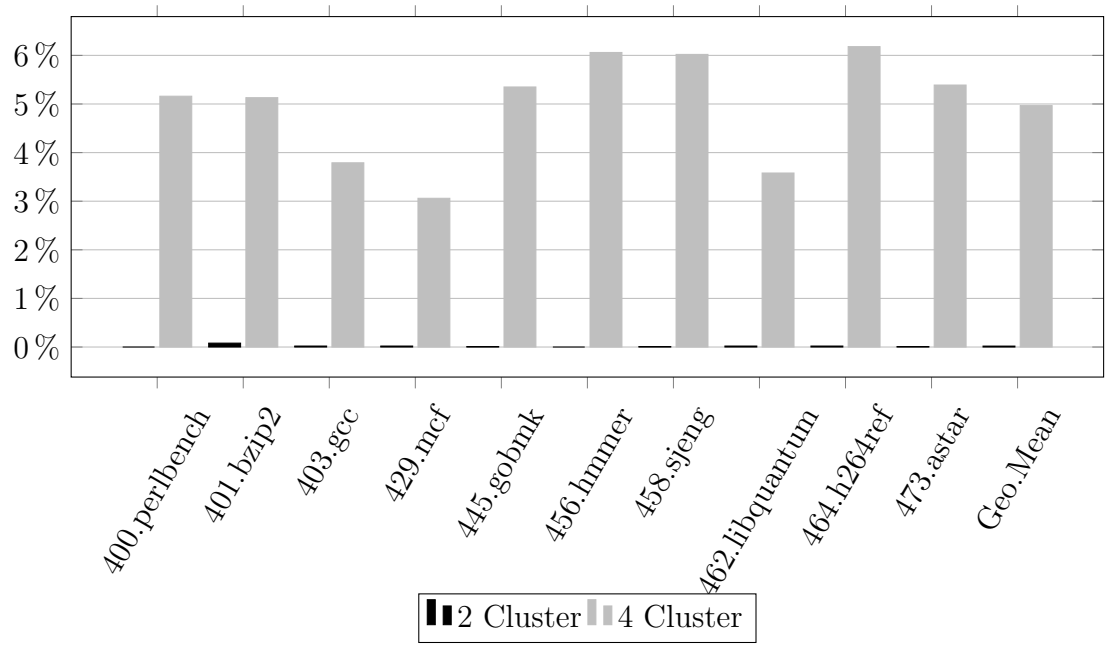


Figure 4.13: Generated Copy Instructions as a Percentage of Total (Unified Window) - Dual Write, Lazy Allocation

small for such a large degree of clustering, 462.libquantum and 464.h264ref still have rather significant IPC loss. As we have discussed before, this is directly related to the number of available registers in each cluster and their sensitivity to extending of the dependence chains. Another factor is the large fan-out from instructions in critical sections of the programs.

In our attempt to keep the total registers constant with respect to the baseline, we leave very few registers in each cluster which can be used for renaming as a significant fraction of these registers hold the in-order state data. This is a very conservative way to assume *constant resources*, which we had been following all along with these experiments. In other words, one can assume that when we divide the register file into

clusters, each cluster can have as many registers as the original register file to keep the access time relatively constant as the process of clustering significantly reduces the required number of ports.

However, reducing the number of ports significantly reduces the area and delay requirements for any memory structure used within a processor. Therefore, another view point in this regard is the total area size, which will be dependent on the number of ports, number of registers in each cluster, as well as the particular technology that can be used to implement the register file. Exploration of the design space with a *constant area* perspective is necessary to optimize a given implementation with respect to the exact number of registers which can be allocated to each cluster. We leave this exploration to future work.

4.9 Clustering the Issue Window and the Register File

While clustering of the register file is important for high performance with reduced complexity, the issue window itself is a significant bottleneck in scaling up superscalar designs. In this section, we explore the impact of clustering the issue window as well as the register file, as shown in Figure 4.3. Such a design has significant advantages in terms of delay, compared to a unified issue window architecture.

When the issue window as well as the register file is divided, we end-up with a

much cleaner design than a unified window. This is because, the issue lanes are private to each cluster, each register file is connected to its own issue lanes and the execution units, and thanks to our novel interconnection mechanism of uni-directional clustering, the wake-up/select logic complexity is also reduced. These benefits however come with a price. In our proposed architecture, the scheduler schedules instructions based on the availability of their registers as well as minimizing the requirements for additional copy instruction generation. This policy inevitably leads to cases where the target issue window that is optimal for scheduling the current instruction is full while a sub-optimal target issue window has available space. Such competition does not occur with a unified window.

On the other hand, it is important to remember that processor performance is not merely IPC. Reduced delays may lead to a better clock rate and the overall performance of the architecture can indeed be better, despite having a lower IPC. Since such an evaluation will require nearly complete designs, in this chapter we continue to rely on the instructions per cycle figures, and conclude that the lower it is the better it will be, if we seek out micro-architecture approaches which reduce complexity. The question then is, how much additional IPC loss do we burden in such a design?

Figure 4.14 shows performance comparison results between the baseline and the clustered architectures where the issue queue is clustered as well as register files, as in Figure 4.3. This is a single-write design and this figure must be compared with

Figure 4.8.

Comparing the two figures, we see about 2% further performance loss in geometric means for both clusters and 464.h264ref jumps to 24% from 20%. Overall, this is expected and the price that needs to be paid for great simplification in routing and wake-up/select, which also should lead to significant power savings due to significantly reduced reservation station complexity.

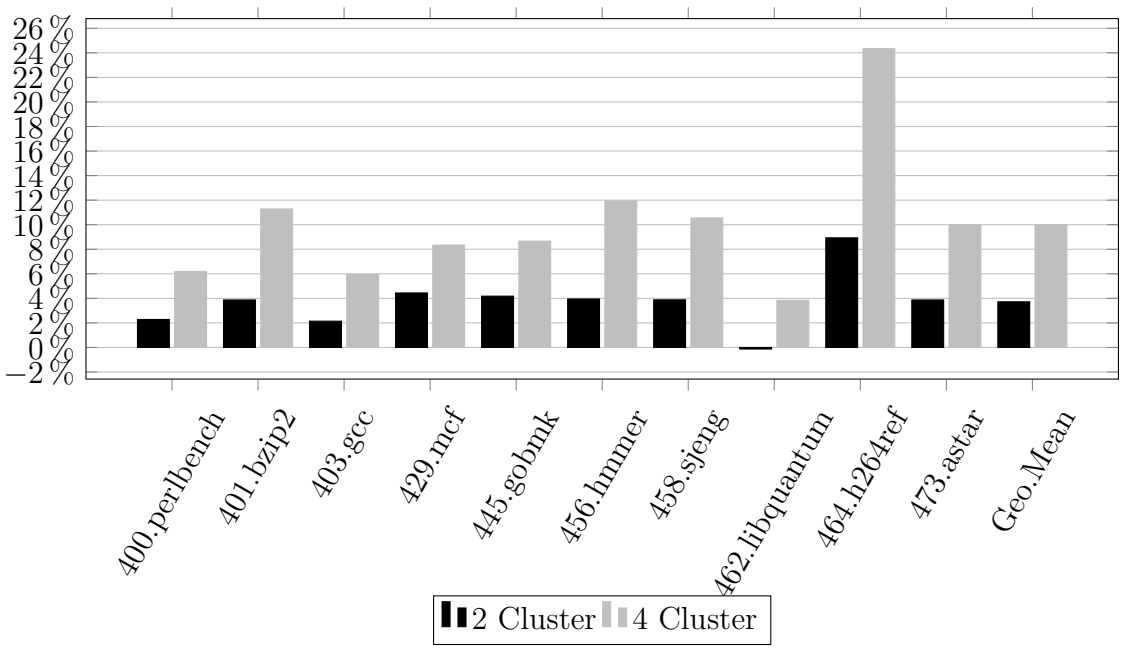


Figure 4.14: IPC Loss when Window is Split (Single Write)

Figure 4.15 shows the ratio of copy instructions generated for this configuration that splits the issue window as well as the register file. These figures are comparable to that of Figure 4.9, indicating that splitting the instruction window does not significantly alter the behavior of the scheduler with respect to copy generation as expected.

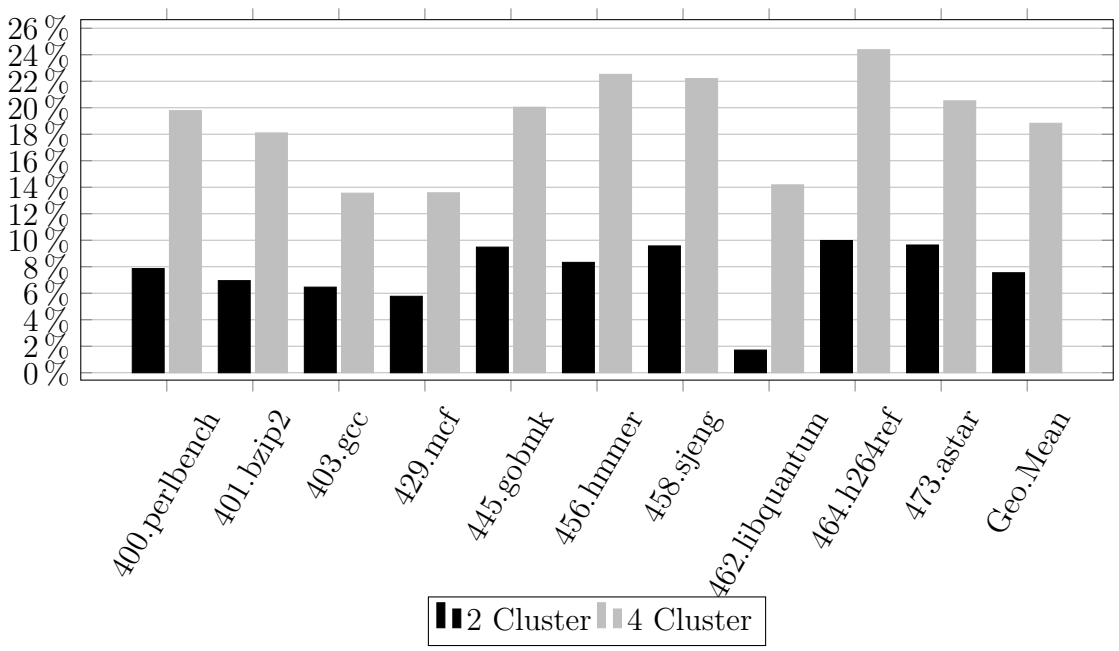


Figure 4.15: Generated Copy Instructions as a Percentage of Total (Single Write)

Clustered issue window and dual-write

Figure 4.16 shows the performance difference between baseline and clustered architectures where the issue queue is clustered and also uses dual-write. Figure 4.17 shows the ratio of copy instructions generated for this experiment.

This experiment shows that combined effects of clustering with the increased register pressure resulting from dual writes is enormous for 462.libquantum, 464.h264ref as well 429.mcf, and the geometric means also illustrate significant IPC loss of up to 10%. This is observed despite having a non-significant change in the number of copy instructions generated, as shown in Figure 4.17.

These graphs clearly indicate that the number of registers available in each cluster

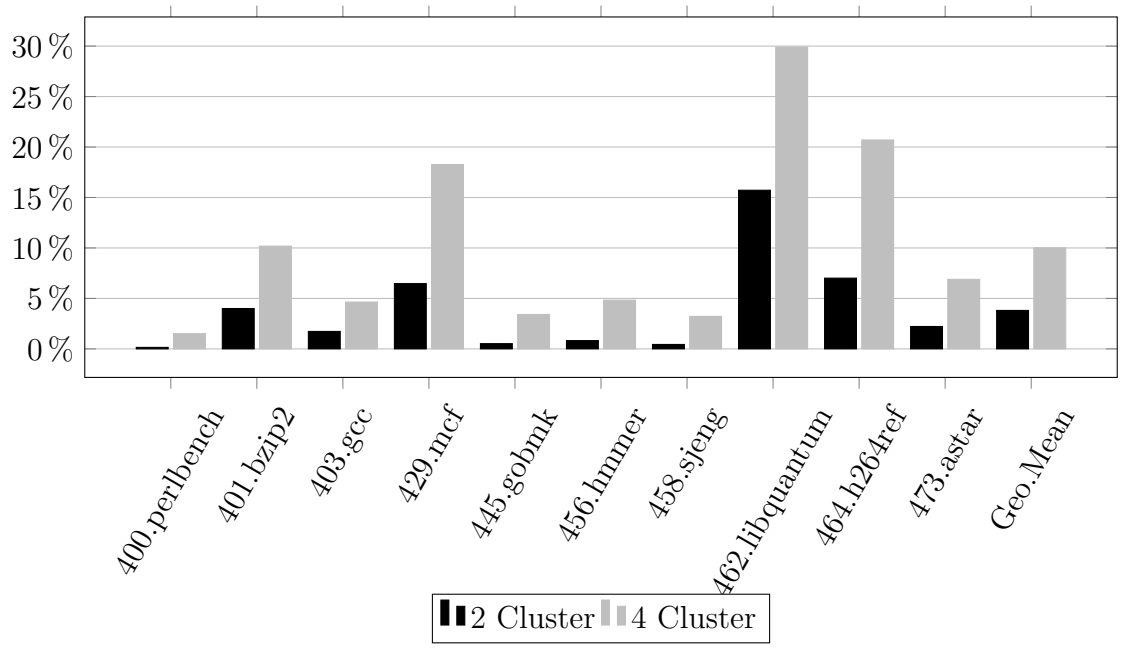


Figure 4.16: IPC Loss when Window is Split - Dual Write

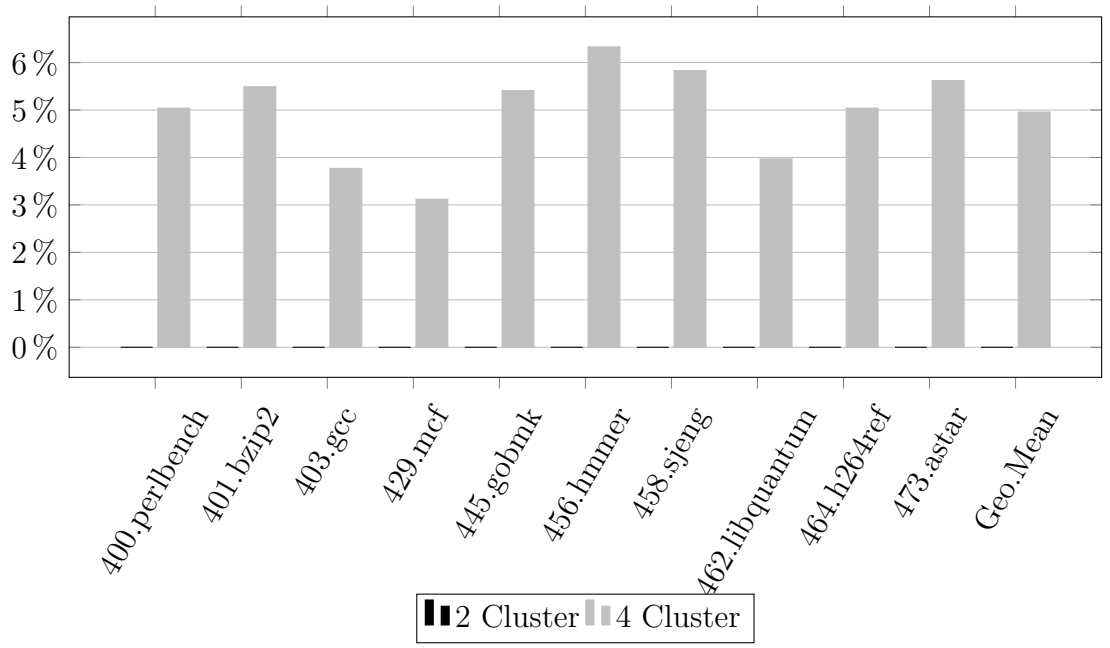


Figure 4.17: Generated Copy Instructions as a Percentage of Total (Split Window, Dual Write)

is a significant factor under this design. Can lazy allocation mechanism remedy the situation when the issue window is clustered as well?

Clustered issue window, dual-write, lazy allocation

Figure 4.18 shows the performance difference between baseline and clustered architectures where the issue queue is clustered and uses dual-write with lazy register allocation. These results indicate excellent performance with the exception of two benchmarks, 462.libquantum and 464.h264ref with geometric means of less than 2% for a 2-cluster and about 5% for a 4-cluster architecture. We do not report the number of copy instructions as they are vey similar to previous configurations.

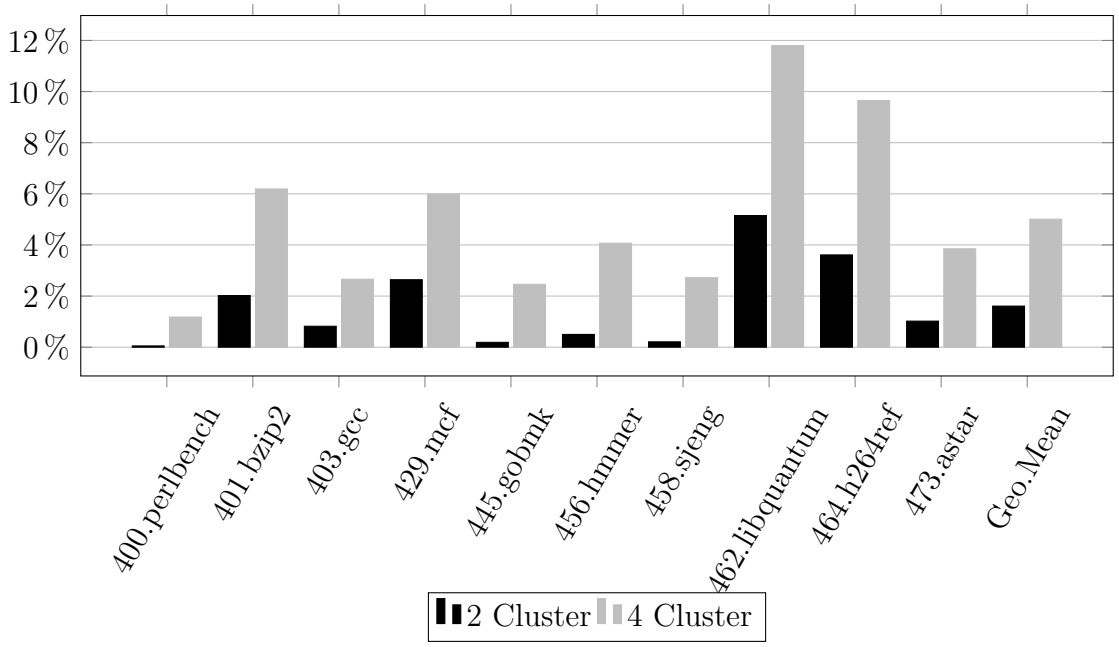


Figure 4.18: IPC Loss when Window is Split - Dual Write, Lazy Allocation

Note that, a two-cluster architecture has an IPC loss of about 5 and 3.8% for

libquantum and h264ref, respectively. These results clearly indicate that with Lazy allocation and dual write, uni-directional clustering offers a competitive and alternative dimension in the design superscalar processors.

4.10 Energy and Power Analysis

Register File Energy Analysis (22nm)						
Used In	**Configuration**			**Dyn. Read Energy (nJ)**	**Dyn. Write Energy (nJ)**	**Total Leakage Power (mW)**
Baseline	320 regs	8R	4W	0.00259572	0.00283489	2.91964
2 Cluster, Single Write	160 regs	4R	2W	0.000839903	0.00129223	1.02726
2 Cluster, Dual Write	160 regs	4R	4W	0.00101048	0.00156153	1.17798
4 Cluster, Single Write	80 regs	2R	1W	0.000473665	0.000609474	0.372189
4 Cluster, Dual Write	80 regs	2R	2W	0.000377798	0.000841728	0.439894

Table 4.3: Per Read/Write Energy Values for Split Register Files

	Read Power	**Write Power**
2 Cluster, Single Write	-65%	-51%
2 Cluster, Dual Write	-58%	19%
4 Cluster, Single Write	-78%	-74%
4 Cluster, Dual Write	-83%	-30%

Table 4.4: Register File Dynamic Power Difference

	Leakage Power Difference
2 Cluster, Single Write	-65%
2 Cluster, Dual Write	-60%
4 Cluster, Single Write	-87%
4 Cluster, Dual Write	-85%

Table 4.5: Register File Leakage Power Difference

We also study the impact of clustering the register file on its power consumption. We used CACTI 7[4][49] to model the register files as SRAM structures. The dynamic read and write energy of various configurations, as well as their leakage power can be seen in Table 4.3. We show that the dynamic read power required for the register files is significantly lower for all configurations, even when accounting for the additional generated copy instructions. We show these results in Table 4.4. The dynamic write power required is also lower in all cases except one. Due to dual-write organizations writing to double the number of registers, the 2-cluster, dual-write organization consumes more power compared to the baseline since the efficiency gained by reducing

the complexity of the register file is not enough to compensate for writing twice the number of registers. Finally, in addition to lower dynamic power consumption in most cases, we also show that the leakage power of the less complex register files are significantly lower as can be seen in Table 4.5.

4.11 Related Work

Palacharla et al. discuss a queue based clustered microarchitecture in [36]. In this clustered architecture, instructions are placed in queues if they are part of the same dependency chain. Therefore, only instructions from the head of each queue could be issued, simplifying the issue logic. Eggers et al. discuss an architecture where the front-end is duplicated, namely a simultaneous multithreading architecture[11], to allow a single superscalar microarchitecture to execute instructions from multiple threads. In contrast, our proposal clusters the back-end of the processor, and is orthogonal to simultaneous multithreading.

The Multicluster Architecture[12] is a close point of comparison to our proposal. The Multicluster Architecture partitions the processor registers as well, and relies on copy instructions but the partitioning is done at the logical level, and requires compiler support to schedule instructions.

Canal et al. discuss repurposing the commonly found intrinsic "clustering" found in most superscalar processors used to divide the integer and floating point functional units [7], and therefore use smaller register files. This work highlights that this

intrinsic clustering works due to the fact that they normally do not share registers, which is one of the observations we exploit in our mechanism.

Balasubramonian et al. explore ways to reduce the complexity of the register file, which includes a multi-level register file organization to reduce the register file size as well as banking to reduce number of ports[3]. Our mechanism achieves both goals by clustering the entire back-end of the architecture. A more efficient way of implementing register banking is also discussed by Tseng et al.[50]. Park et al. approach the issue of reducing register ports and bank conflicts by decoupling the rename stage[37]. This work proposes virtually tagging registers at the front-end of the processor and assigning physical tags only during write-back to avoid instructions retiring in the same group to have bank conflicts.

Chapter 5: Conclusion

In this dissertation, we have introduced the concept of *dependency collapsing* and *dependent parallelism*. This is the first categorization of dependencies according to the method by which they are broken in order to parallelize the bound instructions. Then, we explored two novel techniques of our own to collapse dependencies.

In our first technique, LaZy Superscalar, we present a novel architecture which schedules instructions based on demand signals from other instructions. The architecture naturally eliminates dead code and fuses instructions which can be fused together to remove delays from the critical path of the program. There are several contributions we make:

1. We contribute an alternative superscalar pipeline which implements a general dependency tracking mechanism for all instruction types. We show that such a mechanism is viable for superscalar processors. This general dependency tracking mechanism allows unification of dependence checking for all instruction types and results in a cleaner layout of the pipeline. In this organization, memory operations can be treated very much like any other instruction and can be combined into the main scheduler.

2. We contribute a fine-grain state handling mechanism which permits instructions to be retired in any order and over a large time span. The fine-grain state mechanism should work with other novel approaches in which in-program order of instruction retire may not be possible.

3. We show that aggressive instruction scheduling is not a must for good performance and show that a lazy architecture can be effective in improving performance, particularly with those benchmarks with highly dependent code.

4. We show that a combination of shallow pipelines with operation fusing is another dimension of processor optimization, as opposed to removing delays through deeper pipelining.

In our second technique, we present a uni-directional clustered processor architecture which has not been explored before. We believe that our approach will provide another perspective in designing complex components of superscalar processors, such as the reservation stations and multi-ported register files. A back-end of a processor designed in this manner can also be utilized in a simultaneous multi-threading architecture. Our design relies on a unified front-end. However, the front-end of the processor can also be clustered by simultaneous multi-threading techniques, further breaking apart large structures that would otherwise be needed to avoid introducing resource dependencies.

The *dependency collapsing* classification introduced in this dissertation illustrates that significant performance gains or power savings are possible by collapsing depen-

dencies. Many techniques exist which use dependency collapsing. LaZy Superscalar, in its evolution, started as a dependency collapsing technique that fundamentally targets immutable dependencies. It has developed into the current form described in Chapter 3 where mutable dependencies can also be collapsed, considering existing techniques that collapse mutable dependencies, such as branch prediction and predication. Similarly, the innovation in our uni-directional clustering design comes from exploring techniques that collapse resource dependencies, then extending the idea into its full form as described in Chapter 4.

In conclusion, we show that a better fundamental understanding of the dependency classes introduced in this dissertation can lead to innovation in dependency collapsing techniques. We demonstrate this by the two dependency collapsing techniques we develop, explore and evaluate.

Bibliography

[1] Al-Zawawi, A. S., Reddy, V. K., Rotenberg, E., and Akkary, H. H. Transparent control independence (tci). In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 448–459.

[2] Aşılioğlu, G., Jin, Z., Köksal, M., Javeri, O., and Önder, S. LaZy superscalar. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15* (New York, NY, USA, 2015), ISCA '15, Association for Computing Machinery (ACM), pp. 260–271.

[3] Balasubramonian, R., Dwarkadas, S., and Albonesi, D. H. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34* (Dec 2001), pp. 237–248.

[4] Balasubramonian, R., Kahng, A. B., Muralimanohar, N., Shafiee, A., and Srinivas, V. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Trans. Archit. Code Optim. 14*, 2 (June 2017), 14:1–14:25.

[5] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (May 2000), ISCA '00, ACM, pp. 83–94.

[6] Burger, D., and Austin, T. M. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News 25*, 3 (June 1997), 13–25.

[7] Canal, R., Parcerisa, J. M., and Gonzalez, A. Dynamic cluster assignment mechanisms. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)* (Jan 2000), pp. 133–142.

[8] Cher, C.-Y., and Vijaykumar, T. N. Skipper: A microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 2001), MICRO 34, IEEE Computer Society, pp. 4–15.

[9] Chou, Y., Fung, J., and Shen, J. P. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th International Conference on Supercomputing* (New York, NY, USA, 1999), ICS '99, ACM, pp. 109–118.

[10] Chrysos, G. Z., and Emer, J. S. Memory dependence prediction using store sets. In *ACM SIGARCH Computer Architecture News* (June 1998), pp. 142–153.

[11] Eggers, S., Emer, J., Levy, H., Lo, J., Stamm, R., and Tullsen, D. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro 17*, 05 (sep 1997), 12–19.

[12] Farkas, K. I., Chow, P., Jouppi, N. P., and Vranesic, Z. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1997), MICRO 30, IEEE Computer Society, pp. 149–159.

[13] Fields, B., Bodík, R., and Hill, M. D. Slack: Maximizing performance under technological constraints. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on* (2002), IEEE, pp. 47–58.

[14] Fields, B., Rubin, S., and Bodík, R. Focusing processor policies via critical-path prediction. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on* (2001), IEEE, pp. 74–85.

[15] Gabbay, F. Speculative execution based on value prediction. Tech. rep., EE Department TR 1080, Technion - Israel Institue of Technology, 1996.

[16] Gandhi, A., Akkary, H., and Srinivasan, S. T. Reducing branch misprediction penalty via selective branch recovery. In *Software, IEE Proceedings-* (Feb 2004), pp. 254–264.

[17] Gochman, S., Anati, I., Sperber, Z., and Valentine, R. Fusion of processor micro-operations, 2005. *US Patent* 20040034757 A1.

[18] Gochman, S., Ronen, R., Anati, I., Berkovits, A., Kurts, T., Naveh, A., Saeed, A., Sperber, Z., and Valentine, R. The intel pentium m processor: Microarchitecture and performance. *Intel Technology Journal 7*, 2 (2003), 21–36.

[19] Henstrom, A. Scheduling operations using a dependency matrix, Apr. 29 2003. *US Patent* 6,557,095.

[20] Hilton, A. D., and Roth, A. Ginger: Control independence using tag rewriting. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 436–447.

[21] Hu, S., Kim, I., Lipasti, M., and Smith, J. An approach for implementing efficient superscalar cisc processors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on* (2006), IEEE, pp. 41–52.

[22] Hu, S., and Smith, J. E. Using dynamic binary translation to fuse dependent instructions. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), IEEE Computer Society, p. 213.

[23] Ju, R. D.-c., Lebeck, A. R., and Wilkerson, C. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Ar-*

chitecture (New York, NY, USA, 2001), S. T. Srinivasan, Ed., ISCA '01, ACM, pp. 132–143.

[24] Kessler, R. The alpha 21264 microprocessor. *Micro, IEEE 19*, 2 (1999), 24–36.

[25] Kim, I., and Lipasti, M. Macro-op scheduling: Relaxing scheduling loop constraints. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (2003), IEEE, pp. 277–288.

[26] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (December 2009), MICRO 42, IEEE, pp. 469–480.

[27] Lipasti, M. H., and Shen, J. P. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1996), MICRO 29, IEEE Computer Society, pp. 226–237.

[28] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS VII, ACM, pp. 138–147.

[29] Mahlke, S. A., Hank, R. E., McCormick, J. E., August, D. I., and Hwu, W.-M. W. A comparison of full and partial predicated execution support for ilp processors. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 1995), ISCA '95, ACM, pp. 138–150.

[30] Malik, N. Interlock collapsing alu for increased instruction-level parallelism. In *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on* (1992), IEEE, pp. 149–157.

[31] Moshovos, A., and Sohi, G. S. Speculative memory cloaking and bypassing. *Int. J. Parallel Program. 27*, 6 (December 1999), 427–456.

[32] Önder, S., and Gupta, R. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages* (Chicago, May 1998), pp. 80–89.

[33] Önder, S., and Gupta, R. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages* (Chicago, May 1998), pp. 80–89.

[34] Onder, S., and Gupta, R. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on* (1999), IEEE, pp. 170–176.

[35] Önder, S., and Gupta, R. Load and store reuse using register file contents. In *Proceedings of the 2001 International Conference on Supercomputing* (Sorrento, Italy, June 2001), pp. 289–302.

[36] Palacharla, S., Jouppi, N. P., and Smith, J. Complexity-effective superscalar processors. In *carch24* (June 1997), pp. 206–218.

[37] Park, I., Powell, M. D., and Vijaykumar, T. N. Reducing register ports for higher speed and lower energy. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* (Nov 2002), pp. 171–182.

[38] Perelman, E., Hamerly, G., Van Biesbrouck, M., Sherwood, T., and Calder, B. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev. 31*, 1 (June 2003), 318–319.

[39] Phillips, J., and Vassiliadis, S. High-performance 3-1 interlock collapsing alu's. *Computers, IEEE Transactions on 43*, 3 (1994), 257–268.

[40] Premillieu, N., and Seznec, A. Syrant: Symmetric resource allocation on not-taken and taken paths. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012), 43:1–43:20.

[41] Ronen, R., Peleg, A., and Hoffman, N. System and method for fusing instructions, 2004. *US Patent* 6,675,376 B2.

[42] Rotenberg, E., Jacobson, Q., and Smith, J. A study of control independence in superscalar processors. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture* (Jan 1999), pp. 115–124.

[43] Safi, E., Moshovos, A., and Veneris, A. A physical-level study of the compacted matrix instruction scheduler for dynamically-scheduled superscalar processors. In *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation* (July 2009), SAMOS'09, IEEE Press, pp. 41–48.

[44] Sassone, P., and Wills, D. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on* (2004), IEEE, pp. 7–17.

[45] Sassone, P., Wills, D., and Loh, G. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. *ACM SIGPLAN NOTICES 40*, 7 (2005), 127.

[46] Seznec, A., and Michaud, P. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction Level Parallelism 8* (2006), 1–23.

[47] Sherwood, T., Perelman, E., and Calder, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Com-*

pilation Techniques (Washington, DC, USA, 2001), PACT '01, IEEE Computer Society, pp. 3–14.

[48] Srinivasan, S. T., and Lebeck, A. R. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), MICRO 31, IEEE Computer Society Press, pp. 148–159.

[49] Thoziyoor, S., Ahn, J. H., Monchiero, M., Brockman, J. B., and Jouppi, N. P. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. *SIGARCH Comput. Archit. News 36*, 3 (June 2008), 51–62.

[50] Tseng, J. H., and Asanovic, K. Banked multiported register files for high-frequency superscalar microprocessors. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* (June 2003), pp. 62–71.

[51] Tullsen, D., and Calder, B. Computing along the critical path. Tech. rep., Technical report, University of California, San Diego, 1998.

[52] Tune, E., Liang, D., Tullsen, D. M., and Calder, B. Dynamic prediction of critical path instructions. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture* (2001), pp. 185–195.

[53] Vintan, L. N., and Iridon, M. Towards a high performance neural branch predictor. In *Neural Networks, 1999. IJCNN '99. International Joint Conference on* (Jul 1999), vol. 2, pp. 868–873 vol.2.

[54] Yeh, T.-Y., and Patt, Y. N. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture* (New York, NY, USA, 1991), MICRO 24, ACM, pp. 51–61.

[55] Zaidi, N., Hammond, G., Shoemaker, K., and Baxter, J. Dependency matrix, May 16 2000. *US Patent* 6,065,105.

[56] Zyuban, V., and Kogge, P. The energy complexity of register files. In *Proceedings of the 1998 international symposium on Low power electronics and design* (1998), ACM, pp. 305–310.

Appendix A: ACM Copyright Transfer Agreement

ACM Copyright and Audio/Video Release

Title of the Work: LaZy Superscalar

Publication and/or Conference Name: ISCA '15: The 42nd Annual International Symposium on Computer Architecture Proceedings

Author/Presenter(s): Gorkem Asilioglu:Michigan Technological University;Zhaoxiang Jin:Michigan Technological University;Murat Koksal:Michigan Technological University;Omkar Javeri:Michigan Technological University;Soner Onder:Michigan Technological University

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, or (3) any repository legally mandated by an agency funding the research on which the Work is based.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM templates, you will need to include the rights management and bibstrip text blocks below to the lower left hand portion of the first page. As this text will provide rights information for your paper, please make sure that this text is displayed and positioned correctly when you submit your manuscript for publication.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
Copyright 2015 ACM 978-1-4503-3402-0/15/06...$15.00
http://dx.doi.org/10.1145/2749469.2750409

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government?
○ Yes ◉ No
Country:

II. PERMISSION FOR CONFERENCE TAPING AND DISTRIBUTION (Check A and, if applicable, B)
A. Audio /Video Release

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB

device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately by itself as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? (•) Yes ( ) No

B. Auxiliary Materials, not integral to the Work

Do you have any Auxiliary Materials? ( ) Yes (•) No

I hereby grant ACM permission to serve files named below containing my Auxiliary Material from the ACM Digital Library. I hereby represent and warrant that my Auxiliary Material contains no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software, and I hereby agree to indemnify and hold harmless ACM from all liability, losses, damages, penalties, claims, actions, costs and expenses (including reasonable legal expense) arising from the use of such files.

[ ] I agree to the above Auxiliary Materials permission statement.

[ ] This software is knowingly designed to illustrate technique(s) intended to defeat a system's security. The code has been explicitly documented to state this fact.

III. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

(•) We/I have not used third-party material.
( ) We/I have used third-party materials and have necessary permissions.

IV. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and be sure to include a notice of copyright with each such image in the paper.

(•) We/I do not have any artistic images.
( ) We/I have any artistic images.

V. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

Funding Agents

1. Division of Computing and Communication Foundations award number(s): 1116551, 1450062

DATE: **03/31/2015** sent to galolu@mtu.edu at **12:03:07**