Eager Scheduling of Dependent Instructions

Kurush Kasad
Michigan Technological University, kkasad@mtu.edu

EAGER SCHEDULING OF DEPENDENT INSTRUCTIONS

By

Kurush S Kasad

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2022

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Advisor:	*Dr. Soner Onder*
Committee Member:	*Dr. Zhenlin Wang #1*
Committee Member:	*Dr. Jianhui Yue #2*
Committee Member:	*Dr. David Whalley #3*
Department Chair:	*Dr. Linda Ott*

Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge and thank my advisor Dr. Soner Onder who made this work possible. I started working on my thesis around 18 months ago and this is my first time doing academic research. Dr. Soner guided me throughout the process and helped me during the conceptual, implementation and writing stage. Your kind and friendly nature really helped me keep going during the pandemic.

I would like to thank Dr. David Whalley for providing the necessary compiler optimizations required for the implementation and for your kind and supporting words throughout my thesis. I would also like to thank Dr. Zhenling Wang and Dr. Jianhui Yue for sitting on my panel and taking the time to read my thesis and provide their invaluable feedback. Thank you to all my labmates for their kindness and friendship.

A special thank you to my family and friends for always being by my side - without you, none of this would have been possible.

Abstract

Modern superscalar processors are able to potentially issue and execute multiple instructions per cycle. Several techniques over the years have focused on increasing the Instruction Level Parallelism (ILP) that a processor can exploit. However, there are many limitations of ILP that hinder performance, chief of them being the chain of dependencies between instructions that stops instructions from being executed in parallel.

We propose a new micro-architecture design which extends the superscalar pipeline with a data-flow pipeline where the dataflow part identifies immediately dependent instructions and executes them early. The dataflow pipeline is able to identify redundant instructions, track changes in the operands of the redundant instructions and execute new instructions early in case of operand change. Our design helps alleviate some of the main limitations of ILP.

Chapter 1

Introduction

Superscalar architectures are able to potentially issue and execute multiple independent instructions per cycle. This parallelism which machines exploit at the instruction level is called Instruction Level Parallelism (ILP). Register renaming techniques help eliminate false dependencies in an out of order superscalar and helps in extracting more ILP. A lot of research has been focused on trying to maximize ILP in superscalar processors. However, there are some fundamental limitations of ILP that hinders the processor to utilize ILP to its full power.

One limitation of ILP is related to the size of the issue window. A small issue window does not allow enough independent instructions in it at any given time. Increasing the window size on the other hand leads to hardware complexity which makes it difficult to maintain a high clock speed. Another limitation of ILP is associated with the control flow of a program. Branch predictions in a processor delays the fetching of instructions until the correct target address becomes known. It is not practical to fetch instructions from more than one target address in a single cycle. This delays the filling of the issue window even on a correct prediction.

Another limitation of ILP is associated with data cache misses. Load instructions are often the first instruction in a dependency chain and they often miss in the data cache. These misses translate to the delay of all instructions that are dependent on the load instruction. A non-blocking cache reduces the impact of a data cache miss on ILP but misses still affect ILP performance.

A third limitation of ILP is the inherent sequential portion of computation. The true dependencies in a code can never be parallelized and this sequential component gravely restricts ILP.

Past work has tried to overcome the limitations of ILP by trying to dynamically detect and eliminate redundant computation. We believe that along with eliminating redundant instructions, the processor should also try to execute the next group of immediately dependent instructions. We can achieve a higher ILP by executing dependent instructions earlier which reduces the time to process instructions along the critical path.

We propose a new micro-architecture design which extends the superscalar pipeline with a data-flow pipeline where the dataflow part identifies immediately dependent instructions and executes them early. The dataflow pipeline is able to identify redundant instructions, track changes in the operands of the redundant instructions and execute new instructions early in case of operand change.

The eager execution paradigm includes the following advantages: (1) Redundant instructions are identified in the front end of the processor. Redundant computation is not only eliminated, thereby freeing up resources, but its result is also available earlier leading to execution of its dependent instructions. (2) Immediately dependent instructions are issued and executed earlier, leading to quicker collapse of dependence chains. (3) Early execution of instructions makes their result available to long latency operations earlier, which can improve performace by reducing the length of the critical path.

Chapter 2

Background

Compilers employ a number of optimization techniques to improve code. One such technique is loop invariant code motion. Loop invariant code consists of expressions that can be moved out of a loop body without affecting the program semantics. By moving the invariant code out of the loop, that code only executes once instead of every loop iteration. Another added benefit of code motion is that the result of the invariant code is available to dependent instructions in the loop earlier.

Compiler techniques are able to capture a large amount of static redundancy in computations. However, studies [1] [2] indicate that a large amount of redundancy in programs is actually dynamic.

Sodani and Sohi [3] introduced the concept of Dynamic instruction reuse. They store the results of a previously executed instruction in a hardware structure called the Reuse Buffer(RB) (Figure 2.1).

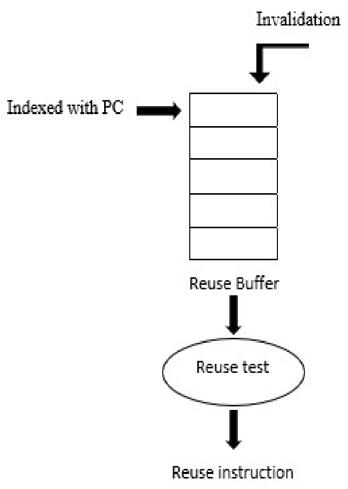


Figure 2.1: Reuse Buffer

After an instruction is decoded, the RB is searched to determine whether a valid result from its previous execution is available. This search is done with the use of the Program counter (PC) of the instruction. The information accessed from the RB then passes through a Reuse Test to determine the validity of the result. Three schemes are presented for reusing instructions. These schemes differ in the way in which the RB results are identified.

The first scheme is based on operand values (Sv). The operand values are stored in the RB along with the result of the instruction. The reuse test in this case simply consists of comparing the source values of the decoded instruction with the values in the RB. A match indicates the result in RB is valid. No explicit invalidation logic is needed in this scheme as a mismatch of operand values during reuse test automatically invalidates the RB entry.

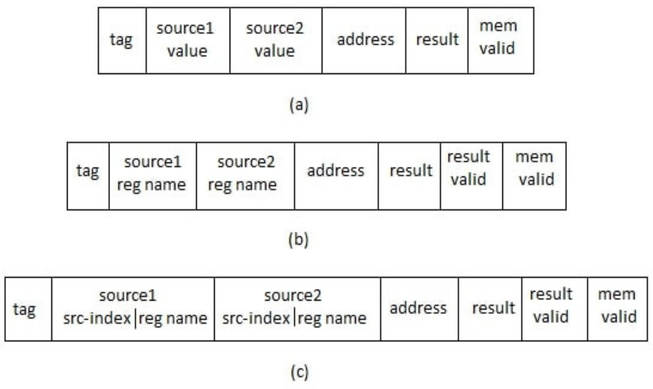


Figure 2.2: $ReuseBufferentry(a)S_v, (b)S_n, (c)S_{n+d}$

The second scheme is based on register names (Sn). In this scheme, the operand architecture registers of an instruction are stored in the RB along with its result. Any writes to architecture registers are broadcasted to the RB for invalidation. The reuse test in this case involves checking whether the corresponding RB is valid.

The third scheme uses register names and dependence chain information to extend Sn (Sn+d). In this scheme, a source index field is added along with the operand register as in Sn. The source index field stores the RB index of the source operand. This creates a dependency chain of the instructions in RB. The reuse test for independent instructions is the same as in Sn. A dependent instruction is valid if source operands (stored in source index) are the latest producers of those registers. Invalidation of independent instructions is the same as in Sn. Dependent instructions are invalidated when their source operands are evicted from the RB.

Richardson [4] observes that long latency operations take up a significant portion of computation time. A result cache is proposed which stores the results of such long latency operations. The result cache is indexed using a hashing of an instruction's source operands. Access to the result cache can be initiated at or before the time of a long latency instruction operation. A hit in the cache makes the result of the instruction available instantly and the already issued instruction can be halted/killed. The execution continues normally on a cache miss and updates the cache after completing execution.

There are a number of key differences between the reuse buffer and the result cache. The foremost difference is that the reuse buffer is indexed with the address (PC) of the instruction while the result cache is indexed with a hash of an instruction's source operands. This difference translates to when the result of a reuse is available to the processor (reuse latency). Access to the reuse buffer can be initiated as soon as the instruction is fetched. On the other hand, the result cache can only be accessed after the instruction has been decoded, renamed and its source operands pass through the hashing algorithm. Thus the reuse buffer makes the result of a reuse available at least one cycle earlier than the result cache. One drawback of the reuse buffer is that since it is indexed with PC, it can only reuse dynamic instances of the same static instruction. A different static instruction with the same source operands (and consequently the same result) will not hit in the reuse buffer. However, since the result cache is indexed with a hash of its source operands, multiple instructions using the same source operands will hit in the result cache.

One advantage of the reuse buffer over result cache is that the entire fetch block can search the reuse buffer in parallel since it is indexed by PC. The dependencies among instructions can then be checked in the same manner as parallel renaming. In case of result cache, parallel lookup of instructions cannot happen since the result of an instruction is needed as the source of another dependent instruction. Thus access to the result cache is inherently sequential and cannot happen in a single cycle.

Molina et.al. [5] proposes the Redundant Computation Buffer (RCB) which seeks to embrace the merits of both reuse buffer and result cache. The proposed RCB has the same reuse latency as the reuse buffer while also being able to identify reuse across different static instructions. On an average, the RCB is able to reuse around 30% of all dynamic instructions.

Yi et. al. [6] observe that some redundant computations in the reuse buffer may be evicted, re-executed and re-stored in the reuse buffer. They go on to say that such computations with a low frequency of execution hurt the effectiveness of storing instructions in a reuse buffer. They introduce a novel approach called instruction precomputation which involves profiling of the program before its execution. Profiling determines the redundant computations with the highest frequencies of execution and stores them in the Precomputation Table (PT) before program execution. The PT is checked during execution to determine a successful instruction reuse. The PT is loaded during the profiling step and does not undergo replacement or eviction of entries. Their approach outperforms similar instruction reuse techniques for similar table sizes while providing a decrease in area, cycle time and port usage.

Several studies have examined the concept of value prediction to achieve a higher ILP. Some of them [7] focus on load value predictions while others [8] extend the concept to predict the values of any instructions that write their result to a register. Value prediction helps in breaking down true data dependency chains by predicting the result of an instruction and allowing the dependent instruction to speculatively execute using the predicted value. Reexecution is necessary in case of a missprediction but many value prediction based techniques have shown a significant increase in ILP.

Value prediction and value reuse capture distinct parts of redundancy in a program. Liao and Shieh [9] propose an architecture which combines the two techniques. They use information from the value prediction table to produce a speculative result from the value reuse table. By combining the two techniques, they are able to achieve a speedup of around 8% over the baseline.

Gellert et. al. [10] observe that branches that depend on dynamic values during execution correspond to a majority of branch misspeculations, even in modern state of the art branch predictors. These branches eat up a lot of cycles during missprediction recovery. They observed that more than 30% branches are dependent on critical load instructions (instructions which miss in the L2 cache) and around 25% of them depend on the result of a multiply or division operation. They postulate that by reducing the latency of these high latency operations, the dependent branches would be executed early and thus reduce the misprediction penalty. To accomplish this, they use a Reuse Buffer for multiply and division instructions. Another table is implemenented which serves as a value predictor for loads that miss in the L1 data cache. By eliminating redundant long latency operations and predicting critical loads, they are able to obtain a speedup of 3.5% in Spec integer benchmarks and around 23% in Spec floating point benchmarks.

Golander and Weiss [11] explore the significance of instruction reuse in checkpoint processors. Checkpoint processors are known for their fast misprediction recovery rate. The recovery process involves two steps: bringing the architecture state back to the last safe checkpoint (rollback), and reexecuting the instruction sequence between the safe checkpoint and the mispredicted instruction. They discovered that a large number of instructions in integer benchmarks rexecute after the state has been restored to the previous checkpoint and a large fraction (nearly 92%) of these instructions already have a result avaiable by the time the misspeculation is detected. They propose that by reusing the isntructions along the reexecution path, a large fraction of redundant computation can be avoided, thus leading to higher ILP.

Huang and Lilja [12] observe that there is a strong correlation between the inputs and outputs of a chain of instructions. They argue that reuse at a basic block level (in contrast to reuse at instruction level) will reduce execution time further while also consuming less hardware. To exploit block reuse, the authors propose a Block History Buffer which stores dynamically determined basic block boundaries along with its inputs and live outputs. The entire basic block is squashed if there is a hit in the block history buffer with a particular series of inputs.

Continuing with the trend to increase reuse granularity, many researches have explored function level reuse. Kavi and Chen [13] replace the entries in the reuse buffer from that of an instruction to that of a function. The reuse buffer is indexed by the PC of the function call and stores the inputs and result of the function. Access to the reuse buffer is initiated at the same time as fetch and a hit in the reuse buffer skips the entire function by correctly changing the PC.

Chapter 3

Exploiting redundancy and its interaction with the fetch engine

There are many parameters that affect the performance of a superscalar. When dealing with code which is highly independent, two parameters become very important to the performance: (a) the number of instructions issued every cycle and (b) number of functional units. Assuming the number of functional units to be greater or equal to the issue width, the IPC of a highly independent program almost reaches the issue width of the processor.

Consider a piece of independent code executing on a 3-wide issue machine with 3 functional units. 3 instructions are fetched every cycle and placed in the issue window.

Assuming no dependencies between the fetched instructions, all 3 instructions can be issued in the issue cycle. Such independent code allows the superscalar to achieve an IPC of 3 (issue width). This scenario is illustrated in Figure 3.1.

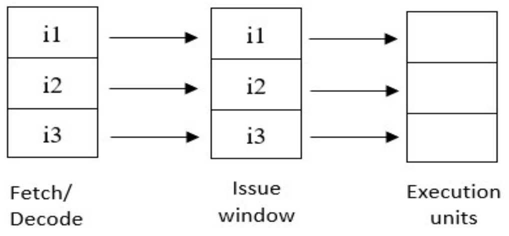


Figure 3.1: Independent code without reuse

Studies have indicated that there exists a large amount of dynamic redundancy in programs. Certain micro-architectures have been proposed which attempt to use this dynamic redundancy to increase ILP. They involve buffering the previous result of an instruction so that future instances of the same instruction can use the result after establishing that the sources of the instruction have not changed. Several of these techniques are discussed in the background section.

Consider the above piece of code executing on a machine utilizing dynamic redundancy. Assume instructions i1 and i3 are buffered in the first cycle. For every other iteration of the code, instruction i1 and i3 are found in the buffer and do not execute. They are essentially squashed as soon as they are decoded. This scenario is illustrated in Figure 3.2.

Dynamic reuse of instructions provides a lot of benefits. The most obvious benefit

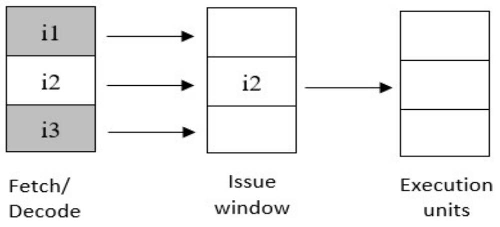


Figure 3.2: Independent code with reuse

is the reduction of resource use in the superscalar. Since i1 and i3 are squashed after being decoded, this frees up the issue window and the execution units and reduces resource contention. Another important benefit of dynamic reuse is that dependent instructions can be executed early since the result of their producer is available early. This helps break down dependence chains sooner and thus increases ILP. Upon evaluation of the code sequence in Figure 3.2, we see that even though instructions i1 and i3 are squashed every iteration, the IPC of the program essentially comes out to 3. Thus dynamic reuse of even two-thirds of the fetch width does not improve ILP.

The solution to increasing ILP with dynamic reuse is to increase the fetch width of the processor. The fetch width was not considered to be an important parameter for increasing performance since a machine without dynamic reuse will not be able to issue more instructions than the issue width. Since there is a possibility of squashing instructions earlier in the pipeline, other instructions from the now-widened fetch group maybe able to proceed to the execution units.

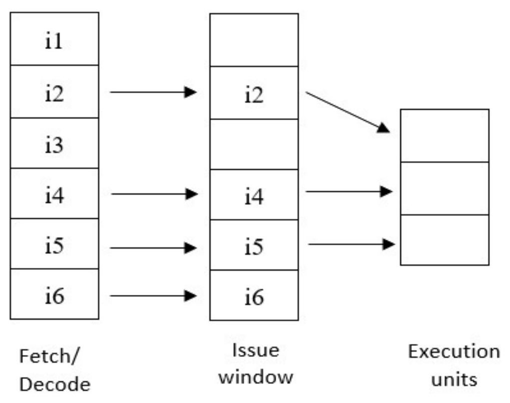
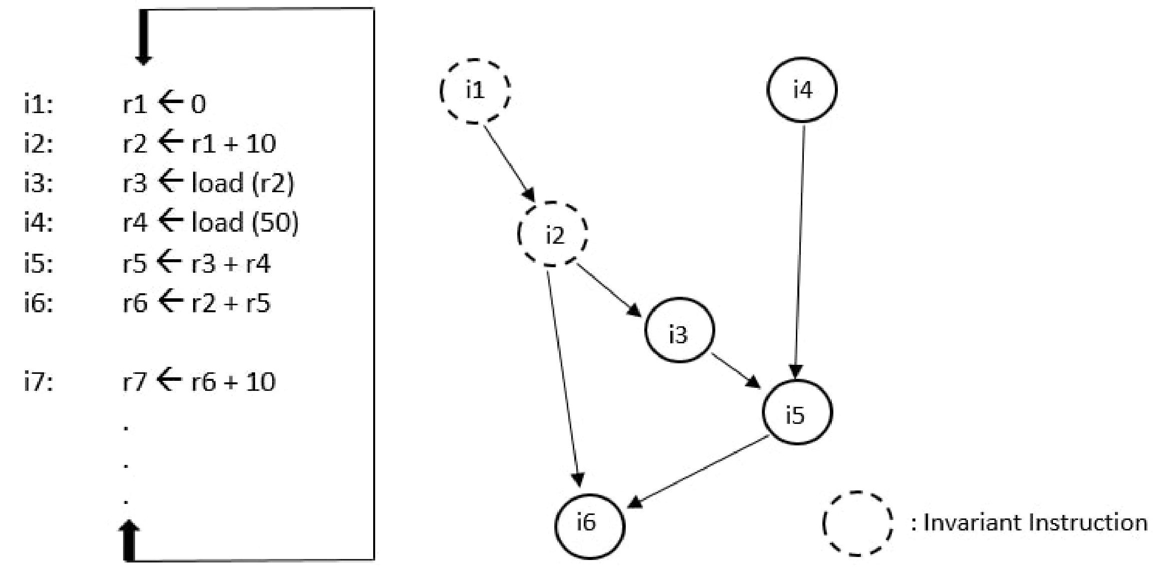


Figure 3.3: Wider front end with reuse

Consider 6 independent instructions being fetched in a 6-wide fetch, 3-wide issue machine. In the first iteration of the loop, instructions i1 and i3 are placed in the buffer and are redundant for all other iterations. The second iteration (and every iteration thereafter) of the loop is illustrated in Figure 3.3. Since i1 and i3 have valid results in the buffer, they will be squashed before they enter the issue window. This allows i4 and i5 to proceed to the execution units one cycle earlier than they would have in the baseline. At this moment of time, the machine with instruction reuse is 2 instructions ahead in its execution than the baseline. This effect is propagated at every instance of an instruction reuse. In a highly independent program, this machine will be able to achieve an ILP of greater than 3 even in a 3 issue superscalar.

The benefits of a wider frontend with instruction reuse are twofold. Since some instructions are squashed at decode, they do not occupy the issue window. This makes space in the issue window for the next fetch group which may contain more independent instructions. The second benefit is that the instructions in the same fetch group which would not have otherwise proceeded to execution are now issued because the issue width has been freed of the instructions which have been reused. Note that the IPC of the baseline with this configuration will never exceed 3.



```
i1:    r1 ← 0
i2:    r2 ← r1 + 10
i3:    r3 ← load (r2)
i4:    r4 ← load (50)
i5:    r5 ← r3 + r4
i6:    r6 ← r2 + r5

i7:    r7 ← r6 + 10
       .
       .
       .
```

(a) Code (b) Dependency Graph

Figure 3.4: Dependent sequence of instructions

A real world program rarely exhibits such high levels of independent code. The ILP of a realistic program is thwarted by the data dependencies between instructions. A realistic code sequence is shown in Figure 3.4(a). The dependency graph of the given code is shown in Figure 3.4(b). (Note: Reuse is only available for arithmetic instructions). Our motivation is to obtain the result of instruction i6 as soon as possible because the rest of the loop depends on the value produced by i6. We can now look at the execution time of these 6 instructions under a narrow and wide front end with and without instruction reuse. Instructions i1 and i2 are cached in the reuse buffer in the first iteration of the loop and can be reused for subsequent iterations of the loop.

The fetch blocks under a narrow and wide front end are shown in Figure 3.5.

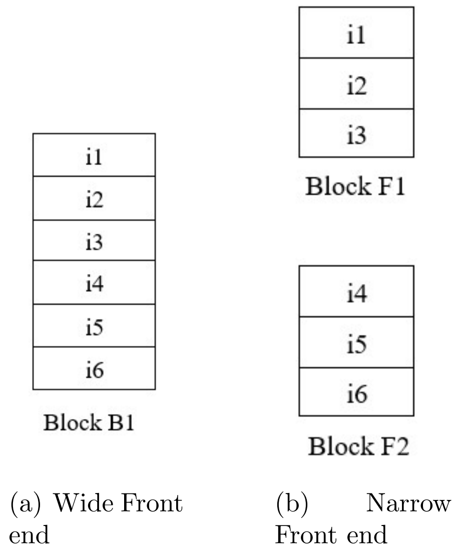
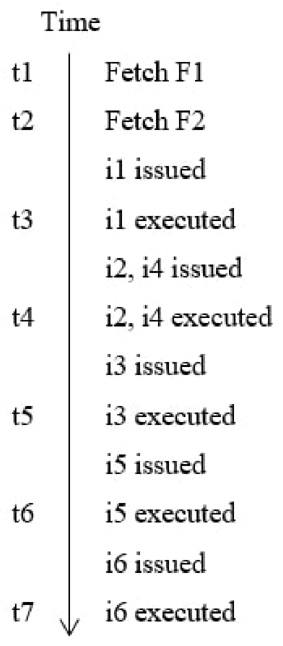


Figure 3.5: Illustration of Fetch Blocks

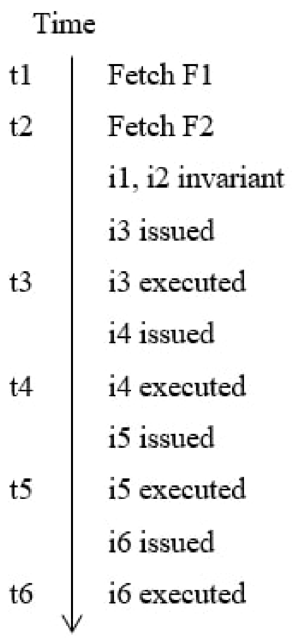
We consider Figure 3.6(a) as the baseline with a 3-wide fetch engine with no instruction reuse. Reusing instructions with a narrow front end (Figure 3.6(b)) increases performance but does not realize the full potential of instruction reuse. In this configuration, i3 is issued one cycle earlier than the baseline since the dependency chain for i3 is collapsed earlier because of reuse of i1 and i2.

Widening the front end without instruction reuse (Figure 3.6(c)) does not offer any benefit in this case because of the existence of dependencies among instructions. A wider fetch engine provides more instructions to the superscalar but dependencies among them are not collapsed fast enough.

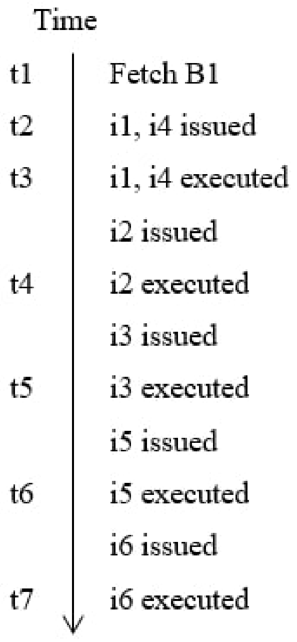
Having a wider front end with instruction reuse (Figure 3.6(d)) with instruction reuse effectively utilizes the fetch bandwidth while also collapsing dependency chains earlier. This case provides the maximum performance improvement among all the cases. Wider fetch bandwidth and instruction reuse complement each other and help increase ILP.



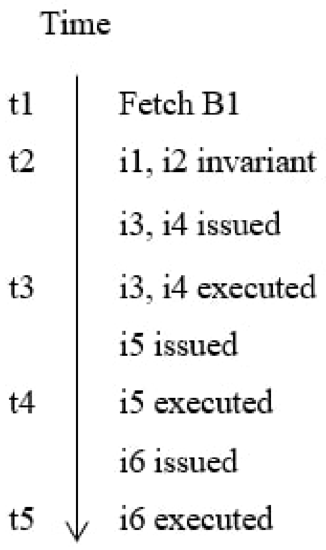
(a) Narrow Front end with no reuse



(b) Narrow Front end with instruction reuse



(c) Wide front end with no reuse



(d) Wide Front end with instruction reuse

Figure 3.6: Instruction Execution (Executions are for second iteration of the loop)

Chapter 4

Eager Execution

Past work has mainly focused on dynamic reuse of instructions which are stored in a buffer to avoid redundant exeuction of instructions. Ideally, the more time an instruction resides in the buffer, the better chance it has of being reused. Several factors limit an instruction from continuing to reside in the buffer - the size of the buffer (older instructions need to be replaced with newer ones when the buffer is full) as well as the limited number of logical locations that the ISA provides and the limited number of physical locations that the hardware provides. Since the ISA has a specific number of logical registers, certain logical names are used multiple times throughout the program. In order to maintain correctness in the buffer, instructions need to be invalidated in the buffer when any of their sources are being rewritten. Since logical names are repeated all the time in programs, the continuous invalidation of instructions in the buffer lead to the destruction of their invariance. An invalidated instruction can no longer be reused since one or more of its source operands have now changed.

Our aim with eager execution is to make use of the invalidated instructions and not discard them as soon as their invariance ends. Henceforth, we will call the buffer which stores instructions as the Eager Shelf. Instead of invalidating an instruction when their source operand is updated in the shelf, we treat the updated operand as a new producer of that instruction. Thus, while finding independent ready instructions to dispatch, we also attempt to execute the next set of immediately dependent instructions from the shelf. The vast majority of instructions encountered in typical programs are operations on one or two operands which are stored in other registers. By eagerly executing instructions whenever their source operand is updated, we increase the ILP as the result of eager execution will be available when the instruction is encountered on the regular path.

The eager execution paradigm combines the power of an out-of-order superscalar with a dataflow style pipeline where the dataflow pipeline makes the capture and early execution of dependent instructions possible. This is accomplised by placing instructions in the eager shelf and re-executing them as soon as any of their producer operands are updated. A speculative dependence graph for the dataflow engine is generated dynamically in the shelf as the superscalar processor keeps fetching new instructions. Instructions are placed in the shelf as they are fetched and their source operands are updated by each new producer instruction writing to the same logical destination. These dependent instructions are dispatched from the shelf as soon as their source operands become ready. The dependence graph built in the shelf is speculative since there is no guarentee that a certain instruction in the shelf will be encountered on the regular path.

In Chapter 1, we discussed the main issues limiting ILP on superscalar processors. One of them is the limited number of reservation stations associated with functional units where an instruction waits for its source operands to become ready. Fully occupied reservation stations stall the fetch engine, thereby halting the ability of the processor to find independent instructions which can be issued. In this case, the eager shelves act as a second set of reservation stations which stores instructions even across control dependencies. An invariant instruction found in an eager shelf does not proceed further in the pipeline. Similarly, a successful hit in an eager shelf for an eagerly executed instruction is not placed in the reservation station since the result of that instruction is already available. Eager shelves decrease the number of fetch engine stalls since instructions found in an eager shelf (whether invariant or eagerly executed) do not occupy a slot in the reservation stations. This approach allows the processor to find independent instructions from future fetch blocks which increases the utilization of execution units which in turn increases the ILP of the program. In some cases, the early availability of results also propagates to branch instructions which are now computed early. Thus eager execution also leads to faster branch computation and as a result, lower branch misprediction delays.

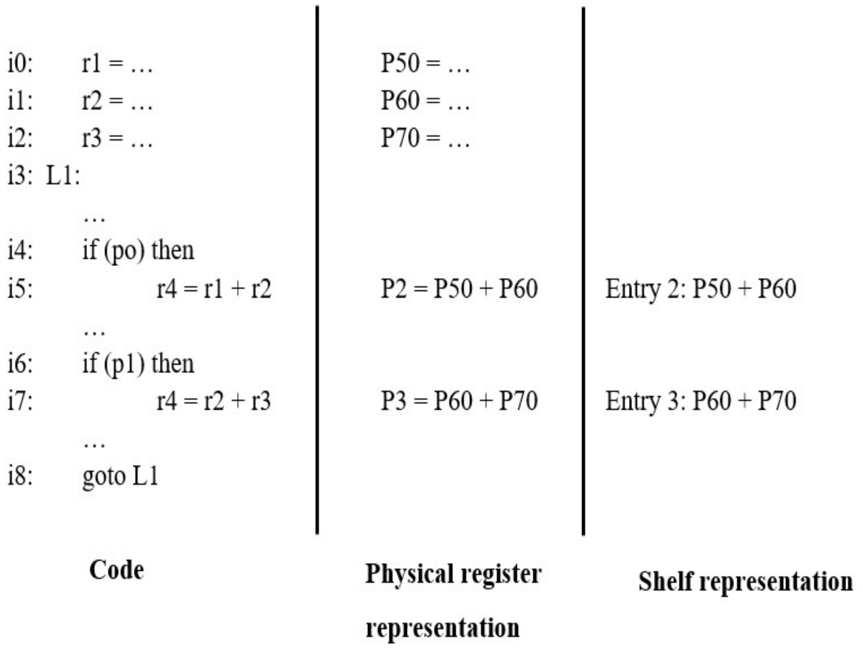
A second factor limiting the ILP is load instructions which miss in the data cache. Generally, loads precede a number of instructions that are dependent on the result of the load. A data cache miss stalls the load instructions and all the instructions dependent on the load. With eager execution, the dependence chain leading to a load is collapsed more quicky, allowing the load to be quickly issued. Therefore data cache misses are triggered earlier and thus the effect of the miss is reduced for future instructions.

A final limitation of ILP is the inherent sequential nature of most programs. The critical path of a program (i.e the height of its dependence graph) is usually confined by the sequential portion of the code. Redundant and eagerly executed instructions help in collapsing the dependence graph. The result of these instructions are avaiable in the shelf and thus redundant computation is either completely avoided or instructions are more quickly executed, increasing the ILP.

4.1 The Eager Shelf

The eager shelf can be thought of as a second reservation station for the dataflow part of the pipeline. Any arithmetic or logical instruction can be placed in the shelf. We will call these instructions as Shelvable instructions. The shelf stores the physical source operands and opcode of Shelvable instructions. The shelf also contains the result of the instruction. The result is stored as a physical register number in the shelf. We design the shelf in such a way that the shelf entry number directly corresponds to the destination physical register of the instruction stored in that entry. Thus an instruction stored in shelf entry 5 will have its result in physical register number 5. This direct correlation allows the shelf, in some cases, to maintain the validity of instructions even when the same logical destination is being over-written. This is because the physical mapping is still retained in the shelf.

Figure 4.1 shows a simple code sequence and how it interacts with the shelf. Instructions i5 and i7 are Shelvable instructions. They are placed respectively at shelf entries 2 and 3. Since shelf entry numbers directly correlate with physical register numbers, these intructions are allocated P2 and P3 as their destination. Both instructions write to the same logical register but their results are still preserved in the shelf. While the map table only maintains the latest mapping, the shelf is able to maintain multiple mappings to the same logical destination. For the next iteration

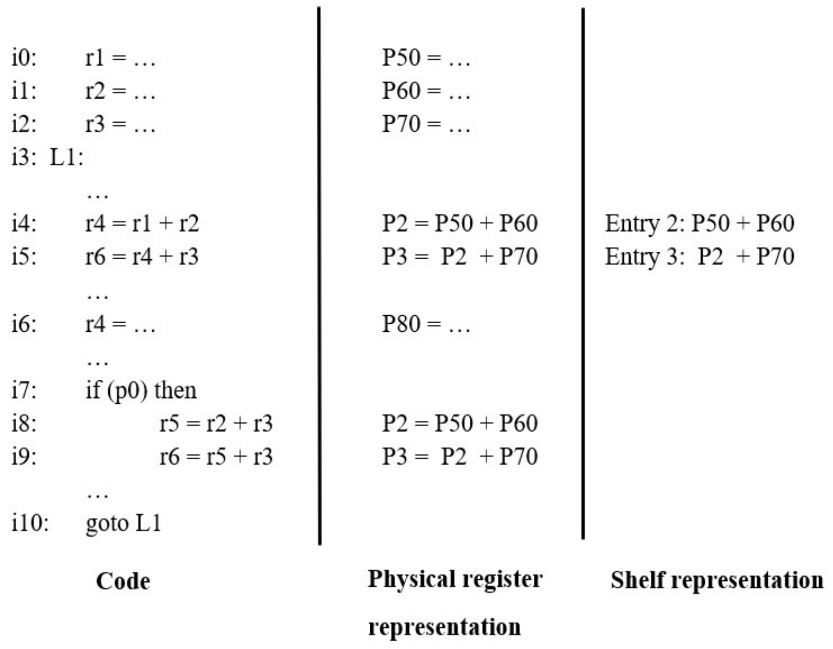


Code	Physical register representation	Shelf representation
i0: r1 = …	P50 = …	
i1: r2 = …	P60 = …	
i2: r3 = …	P70 = …	
i3: L1:		
…		
i4: if (po) then		
i5: r4 = r1 + r2	P2 = P50 + P60	Entry 2: P50 + P60
…		
i6: if (p1) then		
i7: r4 = r2 + r3	P3 = P60 + P70	Entry 3: P60 + P70
…		
i8: goto L1		

Figure 4.1: Code sequence

of the loop, instructions i5 and i7 will search the shelf successfully and the redundant computation will be avoided.

The previous code sequence is now extended in Figure 4.2. In this example, instructions i4 and i5 are placed in the shelf at entries 2 and 3 respectively. One of the sources of the instruction in entry 3 resides in entry 2. As long as entry 2 remains valid, the source operand of entry 3 will remain valid too. This feature allows the shelf to retain a chain of dependent instructions which remain valid in the shelf until the head of the chain is invalidated. By preserving the physical destination of instructions in the shelf and correlating them with shelf entry numbers, we overcome the problem of having a limited number of logical registers in the ISA. Multiple instructions writing to the same logical destination do not invalidate other instructions in the shelf as long as



```
i0:    r1 = …                   | P50 = …           |
i1:    r2 = …                   | P60 = …           |
i2:    r3 = …                   | P70 = …           |
i3:  L1:                        |                   |
       …                        |                   |
i4:    r4 = r1 + r2             | P2 = P50 + P60    | Entry 2: P50 + P60
i5:    r6 = r4 + r3             | P3 =  P2  + P70   | Entry 3:  P2  + P70
       …                        |                   |
i6:    r4 = …                   | P80 = …           |
       …                        |                   |
i7:    if (p0) then             |                   |
i8:            r5 = r2 + r3     | P2 = P50 + P60    |
i9:            r6 = r5 + r3     | P3 =  P2  + P70   |
       …                        |                   |
i10:   goto L1                  |                   |

       Code                       Physical register    Shelf representation
                                  representation
```

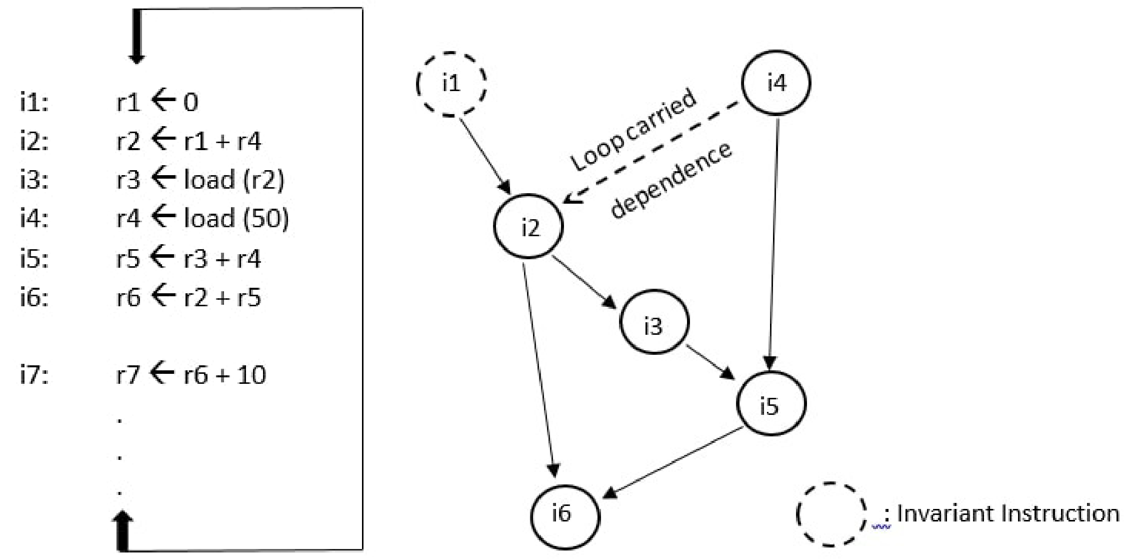
Figure 4.2: Code

their original producers also remain in the shelf.

When the predicate p0 in Figure 4.2 turns true, instruction i8 is searched in the shelf using its physical operands and opcode. The search returns a successful hit at entry 2. Note that instruction i8 has a different logical destination than the instruction placed at shelf entry 2. This distinction does not invalidate the search but actually maps r5 to P2 in the map table. i8 is detected as a redundant instruction and does not execute now. By using physical register identifiers for shelf search, we can map multiple logical registers to the same physical register or the same shelf entry. This is synonymous to Global Value Numbering in compiler literature where underlyiing equivalence is detected regardless of the usage of the logical name space. Because of the mapping of r5 to P2, instruction i9 will also be found in the shelf and will be considered redundant. The shelf is able to dynamically detect equivalence and eliminate redundant instructions just by updating the map table.

4.2 Eager Execution Example

We will now introduce an extra dependence in the example code sequence from Chapter 3 to make the code more realistic and nullify redundancy. The new code sequence and its dependency graph is shown in Figure 4.3. Instruction i2 is made to be dependent on i4 from the previous iteration. This dependence breaks the reusability of i2 since one of its source operands now change at every iteration of the loop.



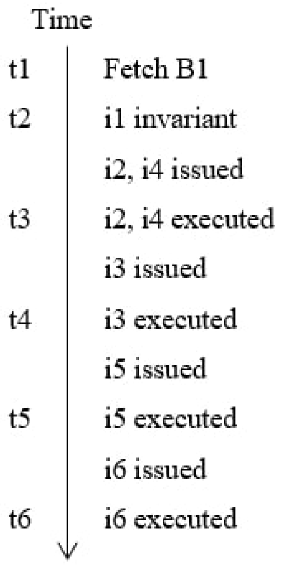
```
i1:     r1 ← 0
i2:     r2 ← r1 + r4
i3:     r3 ← load (r2)
i4:     r4 ← load (50)
i5:     r5 ← r3 + r4
i6:     r6 ← r2 + r5

i7:     r7 ← r6 + 10
        .
        .
        .
```

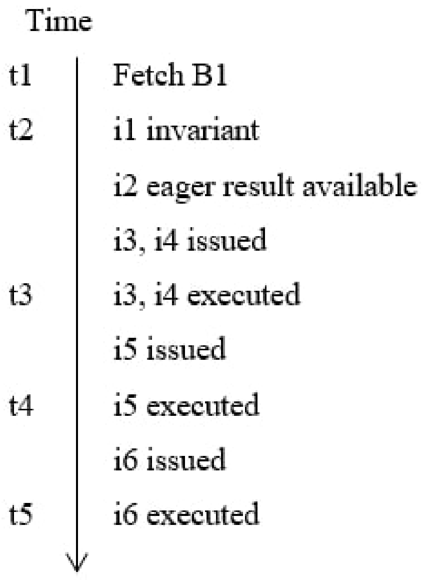
(a) Code (b) Dependency Graph

Figure 4.3: Dependent sequence of instructions nullifying reuse

Instructions i1 and i2 are placed in the Eager Shelf during the first iteration of the loop. Any change in the source operands of instructions in the Eager Shelf triggers eager execution of that instruction. Thus, execution of i4 leads to the eager execution of i2. Note that since eager execution can start as early as the completion of its trigger instruction, i2 in the Eager Shelf begins execution before the second iteration of the loop is even fetched. When the next iteration of the loop is eventually fetched, the result of i2 is already available in the Eager Shelf, making i2 essentially "invariant" to the processor. As shown in Figure 4.4, eager execution beats simple instruction reuse in performance since eagerly executed instructions are additional reusable instructions even with a change of their source operands. (i2 in Figure 4.4(a) can be issued in the same cycle as i4 since i2 is dependent on the previous iteration of i4).



(a) Wide fetch unit with only reuse



(b) Wide fetch unit with reuse + eager execution

Figure 4.4: Instruction execution (Executions are for second iteration of the loop)

Chapter 5

Algorithm

This chapter provides a detailed explanation of our eager execution algorithm.

The algorithm makes use of a shelf structure called the *eager shelf*. The structure is similar to reservation stations in operation, permitting broadcasting, changing the source operands of shelved operations, as well as selecting and issuing ready instructions. In addition to the shelf structure, a free list of available *shelf* entries called *shelf queue* is provided. We also utilize definition and use counters for each *shelf* entry for checking when an entry can be safely released. We begin by describing the organization of the *eager shelf*.

5.1 Eager Shelf

An *eager shelf* entry, although organized similar to a reservation station, requires additional functionality not found in traditional reservation stations. A *shelf* entry needs to be able to accomplish two functions: have a set of identifiers which can uniquely identify an instruction, and have enough information about an instruction to recreate it for eager execution. The former is required to correctly identify whether an instruction has been buffered in the *shelf* while the latter is required to assemble the instruction and send it for execution. By buffering the physical source operands and the opcode of an instruction, the *shelf* can achieve both functions. An instruction can be uniquely identified by the physical source operands and opcode. The execution units for eager execution also need only the physical registers and an opcode for executing instructions.

Apart from the source operands and opcode, the *shelf* also contains additional supporting fields. Each entry contains a *eager* bit which indicates whether the entry is eligible for eager execution. Each entry also has a definition and use count which are meant to check whether an entry can be safely released. The definition count indicates the number of active definitions of an entry and the use count indicates the active use of the entry within the *shelf*. Because we eargerly execute instructions from the *shelf*, the *shelf* needs to have a *ready* bit for each of its source operands, which indicates that the result of the operand is now available. An instruction can be sent for eager execution only after all of its source operands are ready.

A total of $n + m$ physical registers are available to the processor. Physical registers *1:n* are reserved for *eager shelf*, where n is the number of *shelf* entries. The other m physical registers are avaiable in the free register pool. Shelf entry numbers and physical register numbers are directly correlated. For instance, the result of instruction in Shelf entry 5 will be written into physical register 5.

We now describe the operations that can be performed on the *shelf.* Each of these operations support the functioning of the algorithm.

When trying to buffer an instruction in the *shelf*, obtaining an empty entry and placing the instruction are two key operations.

Get free entry: This operation returns an empty Shelf entry by popping from the *shelf queue.*

Place instruction: This operation places a new instruction into an empty entry using its physical source operands and opcode.

With the help of these two operations, we can buffer an instruction in a *shelf* entry. After an instruction has been placed in the *shelf*, the result of that instruction is available to any subsequent iterations of that instruction. To utilize the result, the subsequent instruction needs to search the shelf to check whether a previous version of that instruction was buffered. This is accomplished by the following operation.

Shelf Search: Search the shelf using two register identifiers and an opcode. There can either be one unique Shelf hit or no hit. A hit is termed as *Shelf hit* and no hit is termed as *Shelf miss.*

After an instruction has been buffered in the *shelf*, it remains redundant until there is a change to any of its source operands. The *shelf* needs to be made aware of this change. This is done by the instruction which is writing to the same location. This instructions uses an update operation which informs the *shelf* to change any operands having the old register to the new one. With the updated operand, the *shelf* entry can now be made eligible for eager execution by setting its *eager* bit.

Update-broadcast and send to RS: Update the source operands of existing instructions in the Shelf – thereby invalidating their results and enabling them for eager execution. After the broadcast is complete, the instruction is sent to the reservation stations.

Instructions in the *shelf* with their *eager* bit set are synonymous to instructions in the reservation stations. Both wait in their respective buffers until their source operands are available. As soon as their operands are ready, the instruction can be sent to the execution units. The ready signal for both the *shelf* and the reservation stations arrive from the execution units, which sends the ready signal whenever a result is written into a physical register.

ready-broadcast: Update the ready bit of the source operands in the Shelf – thereby making the shelf entry ready for eager execution.

The select logic checks for all the entries that have its *eager* bit set. Among these entries, the select logic checks for entries which have both of its source operands ready. These entries are selected and sent to the execution units by the following operation.

select: Select an entry that is enabled for eager execution (*eager* bit set) and send it to execution units.

The following operations help in tracking the usage of a *shelf* entry throughout the pipeline. An entry may have been used by either an instruction that is currently in the pipeline or it may have been used as a source operand by another *shelf* entry.

increment def: Every hit in the Eager shelf is a new definition of that entry. The definition counter is incremented to reflect the new definition.

increment use: The use counter of a shelf entry is incremented at every use of the entry's physical register as the source to another shelf entry.

A high level illustration of the pipeline is shown in Figure 5.1. In the next section, we

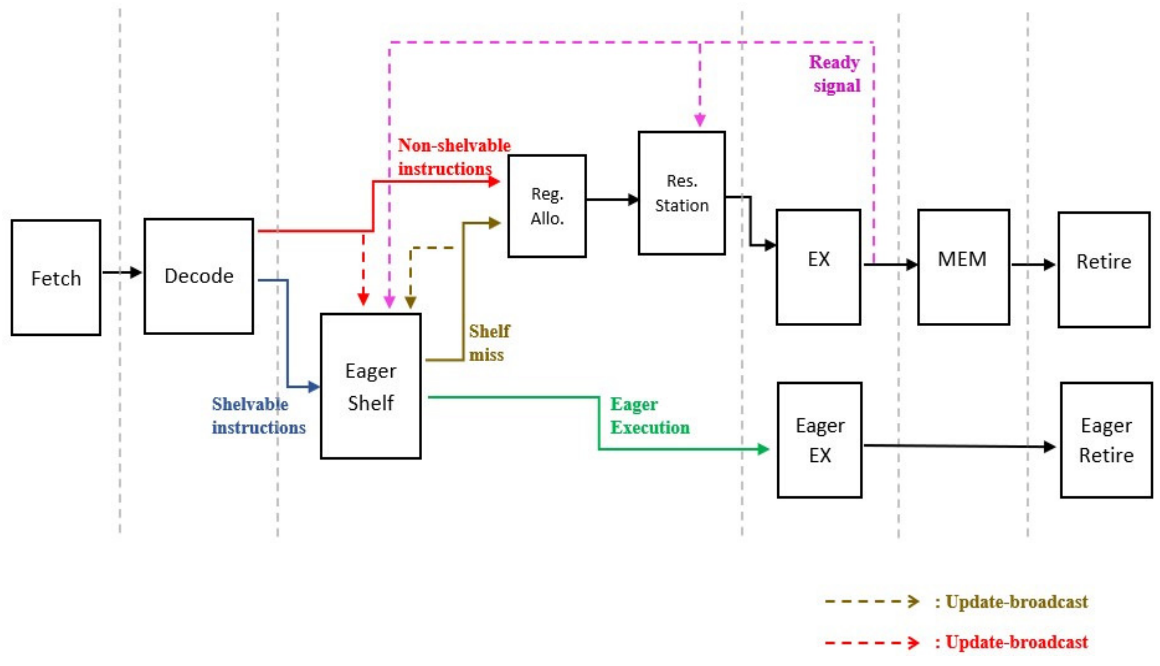


Figure 5.1: High-level illustration of pipeline

discuss the operation of the pipeline by following the instruction flow starting with the fetch stage.

5.2 Fetch Stage

A wider front end is employed in the pipeline to exploit redundancy as mentioned in Chapter 3. The fetch width of the processor will be larger than the issue width. This makes more instructions search the shelf every cycle which would result in a higher reuse count each cycle. On the other hand, this would also fill up the issue window faster. This allows independent instructions to be quickly issued and leads to higher

ILP.

5.3 Shelf Stage

Decoded instructions arriving at this stage are renamed using the map table. Renamed instructions arriving at this point are categorised into 2 groups: *shelvable* and *non-shelvable* instructions.

Shelvable instructions search the shelf using their physical register identifiers and opcode. A successful shelf search indicates that the instruction is either redudant or has been eargerly executed in the Shelf. In either case, the physical destination of the instruction is renamed to the shelf entry number it was found in. This instruction, after updating the map table, is killed at this point. The instruction will not be sent to the issue window. The benefits of this are twofold: the issue window has more empty slots now which can be used by other instructions, and the result of the redundant instruction is already ready which means its dependent instructions can start execution as early as the next cycle.

Shelvable instructions which do not hit in the shelf look for an empty entry in the shelf. The instruction is placed in the empty entry and its destination is renamed to that entry number. Since this instruction did not hit in the shelf, we do not have a result for this instruction. Thus this instruction needs to be executed and is sent to the issue window after updating the map table. Now that the instruction has been placed in the shelf, the next fetch of this instruction will yield a successful shelf hit.

Non-shelvable instructions are renamed as usual by obtaining a free register from the physical register pool. The instruction is sent to the issue window after updating the map table.

The shelf has to be maintained corresponding to the in-order state of the front end of the pipeline. Failure to do so would lead to incorrect shelf hits. Thus each non-redundant instruction needs to update the shelf with its new destination. We accomplish this by broadcasting the previous destination and the new destination of each non-redundant instruction to the source operand field of the shelf. All matches of the previous destination are invalidated and a new entry is allocated in the shelf for that instruction with the new operand. The entry is marked eligible for eager execution. The combination of broadcast and invalidation keeps the shelf updated and synchronized with the map table.

Apart from keeping the shelf synchronized with the map table to keep shelf search operations error-free, we also need to maintain individual shelf entries. Since shelf entries are directly correlated with physical register numbers, the shelf needs to be incorporated with the register release protocol. To accomplish this, we equip each shelf entry with a definition count and a use count. Each successful hit to a shelf entry increments the definition count for that entry. Because we place no limitations on logical to physical mapping, multiple logical registers may increment the definition count of one shelf entry. The definition count keeps track of all active definitions of a particular physical register in the processor. We modify the register release mechanism by preventing any physical register correlated to the shelf to be released if its definition count is not 0.

The definition count monitors the active utilization of a physical register by instructions in the pipeline. We also need to keep track of the number of entries that use the result of one particular shelf entry. To achieve this, we also equip each shelf entry with a use count. The use count records how many other shelf entries use the result of a particular entry (that is, the number of times a particular shelf entry serves as a source operand to other entries). Whenever a new instruction is placed in the shelf, the use count of its source operands are incremented. We again modify the register release mechanism to not release any shelf entries with a use count greater than 0, even if its definition count is 0. This is because while there are currently no active definitions of this entry in the pipeline, there is still one or more instructions in the shelf which use the result of this entry. The definition count and use count, in conjunction, make sure than a shelf entry can never be released when a use of its result is pending, whether in the pipeline or in the shelf.

We now describe exactly how an instruction flows through this stage.

For each instruction i in the rename block:

The source operands of i are renamed from the map table.

If instruction i is ***shelvable***: Search the *shelf* (***Shelf Search:***) using the physical identifiers and opcode. The instruction will either hit in the shelf or miss in the shelf. We describe the operations performed in either case.

Shelf hit: In case of a shelf hit, i is a resuable or eagerly executed instruction. This instruction will not proceed further in the pipeline. The map table is updated to reflect the new physical destination obtained from the *shelf*. Since this is a new definition, increment the definition counter (***increment def***) of this entry.

Shelf miss: In case of a shelf miss, we get the previous destination of i from the map table. Since the instruction was not found in the shelf, we place this instance of the instruction in the shelf. We get an empty entry (***Get free entry***) from the *shelf queue* and ***Place instruction*** in that entry. The shelf maintainance counters (***increment def*** and ***increment use***) are incremented to reflect the change. A new physical register from the free pool is allocated as the new destination. The previous destination and the new physical destination are used to update the source operands in the shelf (***update-broadcast*** *and the instruction is* ***sent to the reservation stations***).

If instruction is ***non-shelvable***: This instruction flows through this stage as an instruction would in a typical superscalar. This instruction does not need to search the *shelf*. We get the previous destination from the map table and allocate a new physical register from the free pool. The map table is updated to reflect the new allocation. The instruction updates the shelf through broadcast and the instruction is sent to the reservation stations (**update-broadcast** and **send to Reservation Stations** (previous dest, new dest)).

As mentioned earlier, each instruction broadcasts its previous destination and new allocation to the shelf. The broadcast is associatively performed and each match invalidates that shelf entry. Since the invalidated shelf entry may not be eligible for release (because of non-zero definition or use counts), we allocate a new entry for the instruction to be copied. The invalidated instruction is copied with the new operand in the empty entry and the entry is marked eligible for eager execution.

5.3.1 procedure *update-broadcast and send to RS(old,new)*

The *old* source operand is broadcasted to the source operand fields of the *shelf*. At every match, we ***get free entry*** to place the new instance of the instruction. The *new* operand is written into the empty entry while all other instruction information is copied from the previous entry. The old entry is invalidated since its source operand was updated. Invalidated entries are not checked during ***Shelf search***. The invalidated entry will remain in the shelf until its definition and use counts are decremented to 0 at which point it will be released and added to the *shelf queue*.

The instruction invoking this procedure is sent to the reservation station and each new shelf entry where the updated instructions is placed is marked eligible for eager execution by setting their *eager* bit.

5.3.2 procedure *get free entry()*

Empty shelf entry numbers are held in the *shelf queue*. If there are shelf entries available in the queue, the procedure pops an entry and returns it.

Each shelf entry has an extra attribute called *age*. Every entry starts out with *age* 0 and its age is incremented by one each cycle until it reaches its *max age*. In case the *shelf queue* is empty (i.e. *shelf* is full), the *shelf* for an entry which does not have a successful hit on it and whose age is *max age* is returned by the procedure.

Note that since this entry does not have a successful hit, the definition and use counts of this entry will be 0. We are sacrificing a potential hit on this entry in the future for the ability to place a new instruction in the entry. The *max age* parameter can be changed accordingly.

5.4 Select Stage

In the select stage, we check all shelf entries and ***select*** entries which are marked for **eager execution** (*eager* bit set) and have their source operands ready, and **send them to execution units**.

Instructions whose source operands are ready in the reservation stations are also selected and sent to the execution units.

5.5 Execute Stage

Instructions are executed and their results are written into their destination register in this stage. ***ready-broadcast*** operation is performed on both the *eager shelf* and the normal path reservation stations. The ready flags of all sources that match with the destination register in both the Eager shelf and the reservation stations are updated.

5.6 Retire Stage

The retire stage in a typical superscalar has two purposes: Update the in-order state of the map table using the destination of the instruction, and release the previous destination of the instruction by adding it to the register pool. All instructions in the eager superscalar follow the first step by updating the in-order map table.

Since some physical registers are now correlated with the eager shelf, we need to modify the register release process. All instructions whose previous destination is independent of the eager shelf (in simple terms - the previous destination number is greater than the eager shelf size), release their previous destination normally - by adding it to the free register pool. All other instructions do not directly release their previous destination. Instead, this instruction will decrement the definition counter of its previous destination. Since the current destination of this instruction has reached the in-order state, the processor can be sure that all other uses of this definition have already retired. However, the eager execution paradigm allows multiple definitions of a single physical register. Thus, we decrement the definition counter instead of directly releasing the previous allocation.

When trying to release a shelf entry (and its corresponding physical register), we look at both the definition and use counts. A definition count of 0 implies that there are no active definitions of that entry in the processor while a use count of 0 implies that no other shelf entry is using the corresponding physical register as its source. The shelf entry can be released only when both of these conditions are satisfied. Upon release, the shelf entry decrements the use count of its physical register.

Chapter 6

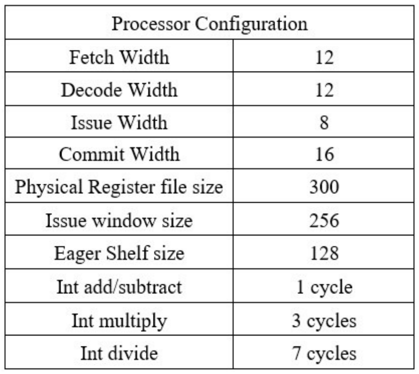
Results and Analysis

6.1 Methodology

We model the eager execution superscalar on ADL [14], an architecture description language which generates cycle accurate simulators. The simulator runs on the MIPS ISA.We use the Spec2006 benchmark suite for performance analysis.

The baseline architecture is a conventional 8-wide issue superscalar with a 12-wide fetch engine. A g-share branch predictor is used in the front end. The complete processor configuration is shown in Figure 6.1.

The baseline architecture and eager superscalar are kept as identical as possible. The



Processor Configuration	
Fetch Width	12
Decode Width	12
Issue Width	8
Commit Width	16
Physical Register file size	300
Issue window size	256
Eager Shelf size	128
Int add/subtract	1 cycle
Int multiply	3 cycles
Int divide	7 cycles

Figure 6.1: Processor Configuration

eager superscalar adds the eager shelf in the front end of the pipeline and execution units in the back end of the pipeline. Instructions executed from the eager shelf have their own set of execution units. This is done to ensure that instructions from the eager shelf do not take up resources meant for normal path instructions.

6.2 Performance Results

We executed Spec2006 benchmarks on the eager superscalar. Figure 6.2 shows the utilization of the shelf as a percentage of total *shelvable* instructions. On an average, only around 20% *shelvable* instructions are placed in the shelf. Since physical registers are correlated with shelf entries, we cannot release a shelf entry until its definition and use counts are 0. In case the shelves are full, incoming *shelvable* instructions will not be placed in a shelf and will be treated as normal path *non-shelvable* instructions. The processor will not be able to take advantage of redundancy of these instructions in case they are encountered again.

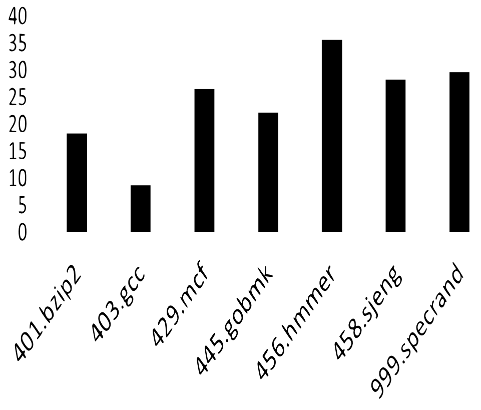


Figure 6.2: Shelf Utilization

Instructions that hit in the shelf can be categorized into 2 groups: Redundant instructions and eagerly executed instructions. Instructions that did not have any change in their source operands from the moment they were placed in the shelf are termed *redundant* instructions or *reused* instructions. Instructions with an updated source operand and issued from the shelf early are termed *eargerly executed* instructions. Figure 6.3 shows the percentage of reused instructions and eagerly executed instructions for all hits in the shelf. Programs with dependent instructions close to each other will have more reusable instructions than eagerly executed instructions. This is because instructions in the shelf need at least a cycle to eagerly execute after their source operands have changed. Dependent instructions in the same fetch group or the immediate next fetch group will not be able to take advantage of eager execution since the shelf will not have had enough time to start eager execution. On the other hand, dependent instructions in different fetch groups will have a better chance of being executed early. The source change by the producer will trigger eager execution. By the time the dependent instruction is fetched, the shelf will have its result ready.

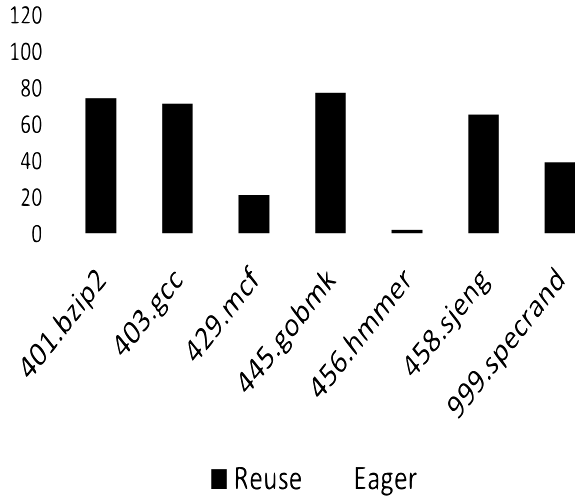


Figure 6.3: The percentage of redundant instructions and eagerly executed instructions

Table 6.1 shows the decrease in the number of cycles for each benchmark.

Benchmark	Baseline cycles	Decrease in number of cycles
401.bzip2	4,952,927,743	38,210
403.gcc	2,316,538,235	45,605
429.mcf	2,076,526,594	350,766
445.gobmk	387,013,197	21,030
456.hmmer	1,023,002,150	4,272,094
458.sjeng	7,768,552,457	1,854,333
999.specrand	35,623,757	19,351

Table 6.1
Decrease in number of cycles for Eager superscalar

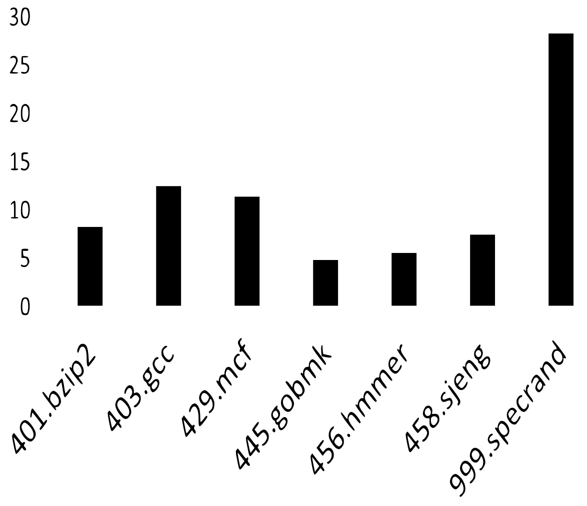


Figure 6.4: Total shelf hits as a percentage of number of instructions placed in shelf

6.3 Analysis and Future Work

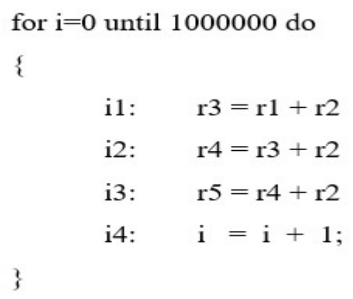
We do not see a substantial decrease in the number of cycles for most benchmarks. This is, in part, because of the low utilization of the shelf. Since we only average around 20% utilization, we cannot take advantage of the other 80% of instructions that may have been reused. Because of low utilization, the shelf may not be able to buffer critical path instructions too. Figure 6.4 shows the percentage of instructions that hit in the shelf to the total number of instructions placed in the shelf. We see around 10% hit rate on 20% utilization which suggests that a very small number of resuable instructions are actually being reused.

There are a number of improvements that can be made on top of the eager execution paradigm. For our simulator, we consider only single cycle arithmetic and logical instructions as *shelvable* instructions (instructions that can be placed in the shelf). By changing the design of the eager shelf, one can easily place multiply and division instructions in the shelf. These instructions are multi-cycle instructions which can provide a boost in performance if they are reused or eagerly executed. The downside of eagerly executing these instructions is that every unsuccessful early execution eats up many cycles of an execution unit.

Another category of instructions that can be placed in the shelf are load instructions. A new paradigm would need to be established in the shelf which would keep track of the dependencies between loads and stores. The shelf will be treated as the top level cache in case of a load reuse while early execution of a load will essentially act as a prefetch mechanism.

We can also program the compiler to use a certain set of logical registers for memory instructions. This will make sure that arithmetic instuctions independent of the load do not invalidate the loads in the Shelf. Instructions writing to a register belonging to the memory set will be the only instructions responsible for invalidating and eagerly executing load instructions which will reduce wasteful early executions.

In our design, we effectively clear the shelf at every branch misspeculation. This is not a necessary requirement. We propose two methods which can be explored in the future with regards to maintaince of shelf during a misspeculation: in an architecture using reorder buffer, the instructions from the tail of the reorder buffer to its head can use their physical destination registers to invalidate any matches in the shelf. This will not invalidate any entries which were placed before the misspeculation and these entries can be reused after the recovery process ends. The second method is motivated from the concept of checkpoint processors. A checkpoint of the eager shelf can be taken at every branch or at a regular interval of cycles. Upon a misspeculation, the last safe checkpoint is copied to the shelf. The ILP immediately following a misprediction is very low in typical programs and having redudant results available in the shelf will lead to a boost in the ILP.



```
for i=0 until 1000000 do
{
        i1:     r3 = r1 + r2
        i2:     r4 = r3 + r2
        i3:     r5 = r4 + r2
        i4:     i  =  i  +  1;
}
```

Figure 6.5: A simple for loop

Certain complier techniques may also be able to aid in eager execution. Consider a simple for loop that runs for a million iterations as shown in Figure 6.5. The shelf, by virtue of correlating physical register and shelf entry numbers, is able to capture the entire dependency chain between instructions i1, i2 and i3. All three instructions will be considered redudant instructions for all iterations of the loop except the first. However, instruction i4 serves as a bottleneck for this loop. i4 creates a million instruction long sequential dependence chain. Although the shelf eliminates every instruction in the loop, we see almost negligible gain in performance. By using some compiler techniques like loop unrolling, we may be able to reduce the impact of i4 on the performance. This loop unrolled by a factor of 10 will see almost a 10-fold decrease in execution time since every instruction apart from i4 will be eliminated because of redundancy.

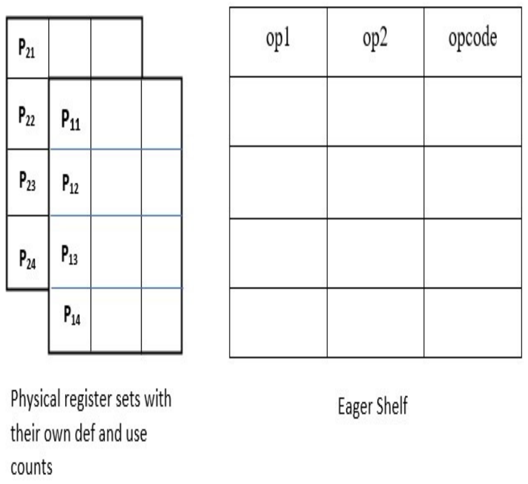


Figure 6.6: Eager shelf with multiple register sets for each shelf entry

As seen in Figure 6.2, the utilization of shelf is not even 50%, even for a shelf size of 128. Increasing the shelf size is counterproductive, since searching the shelf will take more time, leading to an increase in the clock speed. Increasing shelf size also leads to an increase in the area of the shelf. Instead of adding more shelf entries, we can provide more physical registers to each shelf entry. Instead of each entry being correlated with one physical register, now each entry is correlated with 2 physical registers as shown in Figure 6.6. Thus, each entry will have its own set of physical registers. Each set of physical registers has its own definition and use counts. When an instruction is placed in an entry, one of the registers from the set is allocated to that entry. The definition and use counts for that register are updated per the algorithm. When the shelf is full, we empty an entry for the new instruction but do not necessarily release the register associated with that entry. This register will be released only when its definition and use counts reach 0. Instead, the register from the other set is allocated to the new instruction. In this way, multiple registers can be provided for each shelf entry. We can increase the utilization of the shelf without increasing the shelf size by a sizeable amount.

Chapter 7

Conclusion

Eager execution is a novel idea that builds upon the work of instruction reuse. Many instruction reuse techniques have been proposed in the past but all of them destroy invariance with a change in source operands. Our work utilizes the changes in source operands and treats it as a potential future producer to start eager execution. Eager execution helps eliminate dynamic redundancy and helps collapse dependency chains earlier even in highly sequential programs.

Our preliminary work has shown small but promising improvement in ILP with the help of eager execution. We believe that there is a large scope for improvement in this paradigm and even better improvement in ILP in the future.

References

[1] Sodani, A.; Sohi, G. S. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 35–45, New York, NY, USA, 1998. Association for Computing Machinery.

[2] Yi, J.; Lija, D. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 74–81, 2001.

[3] Sodani, A.; Sohi, G. S. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, page 194–205, New York, NY, USA, 1997. Association for Computing Machinery.

[4] Richardson, S. E. Caching function results: Faster arithmetic by avoiding unnecessary computation Technical report, USA, **1992**.

[5] Molina, C.; González, A.; Tubella, J. In *Proceedings of the 13th International*

Conference on Supercomputing, ICS '99, page 474–481, New York, NY, USA, 1999. Association for Computing Machinery.

[6] Yi, J. J.; Sendag, R.; Lilja, D. J. In Monien, B., Feldmann, R., Eds., *Euro-Par 2002 Parallel Processing*, pages 481–485, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[7] Lipasti, M. H.; Wilkerson, C. B.; Shen, J. P. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 138–147, New York, NY, USA, 1996. Association for Computing Machinery.

[8] Lipasti, M.; Shen, J. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237, 1996.

[9] Abstract exploiting speculative value reuse using value prediction. hung Liao, C.; jiann Shieh, J.

[10] Gellert, A.; Florea, A.; Vintan, L. N. *J. Syst. Archit.* **2009**, *55*(3), 188–195.

[11] Golander, A.; Weiss, S. In *Transactions on High-Performance Embedded Architectures and Compilers II;* Springer-Verlag: Berlin, Heidelberg, 2009; page 242–268.

[12] Huang, J.; Lilja, D. J. *Proceedings Fifth International Symposium on High-Performance Computer Architecture* **1999**, pages 106–114.

[13] Kavi, K. M.; Chen, P. 2003.

[14] Onder, S.; Gupta, R. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*, pages 80–89, 1998.

[15] Richardson, S. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 220–227, 1993.

[16] Önder, S.; Gupta, R. In Sakellariou, R., Gurd, J., Freeman, L., Keane, J., Eds., *Euro-Par 2001 Parallel Processing*, pages 418–427, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.