Speculative Super-optimization

Dynamic Binary Vector Widening

A thesis presented for the degree of
Master of Science

by

Conner Ward

Department of Computer Science
University of Virginia
United States of America

December 2022

Abstract

General purpose computer architects are tasked with designing processors that are both fast and useful for a wide variety of program workloads, but it's often difficult to achieve both. This thesis identifies the vector processing unit of the x86 architecture as a valuable hardware resource that produces substantial performance improvements over scalar code, but also a resource that is often difficult or even impossible to leverage to its fullest. Modern high-performance x86 processors support vector widths up to 512-bits, but this capability is underutilized by legacy vector codes as well as target-flexible vector codes that have been compiled to a smaller vector width. Additionally in heterogeneous architectures, the largest vector widths that are implemented in the high-performance cores are often disabled to allow all cores on the processor to share the same instruction set.

This thesis seeks to address this under-utilization of the vector processing unit by speculatively widening vector instructions. The motivating observation is that vector loops whose memory accesses are strided to the vector width can be dynamically unrolled, whereby any vector instructions can then be fused together into a single instruction that uses a multiple of the original vector width. To explore this idea to its fullest, all the necessary algorithms and structures to perform the analysis, transformation, and speculative issue of dynamically widened vector instructions are implemented in the gem5 microarchitectural simulator for the x86 instruction set architecture. This implementation is realizable in hardware, adds minimal overhead to the processing pipeline, and achieves near best-case performance gains for a specific type of vector loop. While the vector codes that can be safely widened with this method are limited, this work displays the potential for similar highly aggressive speculative binary transformations.

Acknowledgements

My sincerest thanks go out to my thesis advisor, Professor Ashish Venkat, and PhD student Logan Moody. It was only with their guidance and extensive microarchitecture knowledge that this thesis was fully realized, and many of the original contributions of this work were formalized in our discussions as a team.

Personally, I want to thank my partner Erin, our dog Turbo, and my parents. This thesis has been the most substantial and challenging project I have ever undertaken, and I owe its successful completion largely to their love and support.

- Conner

Contents

Chapter 1

Introduction

Microarchitecture is the innovation and organization of the various logical and structural components of a processor to implement a given instruction set architecture as optimally as possible. While optimality can be weighted towards a variety of different measures - power consumption, execution latency, heat output, implementation overhead, fault tolerance, or security guarantees - microarchitects always strive to avoid wasting resources. General purpose processors will always require overhead to implement complex instruction sets, and many microarchitectural optimizations create waste in the pursuit of aggregated performance gains, but devoting valuable die area to transistors that are not able to be leveraged by a significant portion of user programs is a waste that is difficult to justify. One such common microarchitectural feature that enables high levels of performance but is not easily leveraged by programs is the vector processing unit.

The vector processing unit in high performance processors exploits data-level parallelism in code by allowing the same instruction to be run on multiple data elements simultaneously. The number of data elements that a vector processor can operate on in parallel is a function of the data element size and the vector width supported by the processor. For example, a processor that implements 128-bit wide vectors can operate on anywhere from two 64-bit data elements to sixteen 8-bit data elements. Therefore, the speedup that vector codes obtain over equivalent scalar codes is defined by the same function of maximum vector width supported by the processor and data element size. As a result, implementing larger vector widths in the processor allows programmers to maximize the data level parallelism found in their codes and generate the greatest levels of speedup.

1.1 Motivation

Despite the higher speedup potential offered by larger vector widths in vector processors, there are both physical and functional barriers that prevent the widespread use of these larger widths. In this thesis, we focus on Intel's design of the x86 microarchitecture as their processors are ubiquitous in the high-performance computing landscape [1]. More importantly, we identify two distinct obstacles that prevent many user programs from fully leveraging the largest vector widths supported in the x86 microarchitecture.

First, vector instructions in x86 assembly code are specific to a given vector width. This means that the x86 instruction that performs an operation on 128-bit vector operands is distinct from the instruction that performs the same operation on 256-bit vector operands, and this is true for each supported vector width and instruction operation. As a result, a vector code can only take advantage of a given vector width if it is compiled to that exact width. For vector codes that were handwritten by programmers for a specific vector width, it would be a substantial undertaking to rewrite all of that vector code for a newly supported and larger vector width. This problem is exacerbated for legacy vector codes that are no longer actively developed or maintained. Even for scalar codes that were automatically vectorized by the compiler or a similar optimization tool, the source code would have to be recompiled to take advantage of a larger vector width that a given processor supports. Both of these processes are costly and discourage programmers from undertaking the efforts required to increase the vector width of their code. In addition, different processors support different vector widths, making any vector code that needs to run on a variety of processors bounded by the greatest common vector width supported.

Second, there has been a substantial increase in popularity of heterogeneous processors in recent years. Heterogeneity in microarchitecture refers to the incorporation of different cores of variable capabilities on the same processor. These designs have the benefit of being able to align the program requested to run on the processor with the core(s) that would most optimally execute that program, but introduce significant challenges in instruction set design. The simplest approach to ensuring correct execution of programs on heterogeneous processors is to require that all cores share the same instruction set. In a heterogeneous design where the cores support different vector widths, those cores that have unique vector capabilities will have their associated instructions removed from the instruction set and that capability disabled in hardware. Intel's most recent heterogeneous architecture, Alder Lake, suffers from exactly this problem [2]. These designs display a clear and obvious waste of hardware resources; a portion of a core's implementation has been designated to supporting

large vector widths, but it is impossible for a program to leverage this functionality.

The net effect of both of these obstacles is that many vector codes that display a high level of data parallelism are often not able to take advantage of the full vector width supported by the processor. This thesis introduces a novel optimization technique that directly addresses this issue.

1.2 Thesis Objective

This project's overarching goal is to enable vector programs to utilize the largest vector width supported on the processor, regardless of the vector width of the compiled code and without needing any help from the programmer or compiler. To achieve this goal, we propose dynamically widening vector instructions in the processor. We specifically target loops that contain vector code and increment the loop induction variable by the vector width each iteration. If vector memory accesses are based on this same induction variable, we observe that the vector instructions across every pair of two loop iterations can be fused into an equivalent vector instruction of double the original vector width. In the optimal case, this transformation cuts in half the number of vector instructions issued to the processing pipeline in a given vector loop and results in a more efficient utilization of the processor's vector hardware.

This thesis begins by providing background on various microarchitectural and programming concepts that are frequently referenced throughout the succeeding sections. Chapter 3 discusses three works directly related to dynamic binary vector widening.

With all the necessary context provided at this point, Chapters 4 and 5 dive deep into the details of vector widening and contain many of the original contributions of this thesis. Chapter 4 begins with a detailed example of vector widening, which in turn reveals why this transformation needs to be issued speculatively in the processor. The final section then examines a more complicated vector widening example, which provides valuable scope to our later work. Chapter 5 contains a technical overview of everything that is required to implement vector widening in an actual processing pipeline. Everything from the high-level algorithms and structures to the hardware-specific additions required of vector widening are explained in great detail.

Chapter 6 covers our experimental methodology, and in particular, all of the work that was done to actually implement vector widening in the gem5 microarchitural simulator [3]. This methodology is put to the test in Chapter 6, which highlights both the immense potential and limitations of vector widening. Finally, this thesis closes with a brief discussion of future research directions and a summarizing conclusion.

Chapter 2

Background

This chapter provides a brief overview of various topics that will aid the reader in following the remainder of the thesis.

2.1 x86 Vector Support

The x86 instruction set architecture first supported vector processing, or single-instruction multiple-data (SIMD) processing, with Intel's MMX technology in 1997 [4]. While MMX only implemented vector instructions for integer data types, used a relatively small vector width of 64-bits, and reused the preexisting floating point registers to hold vector registers, the technology established many of the naming conventions and design patterns seen in all future Intel designed SIMD instruction set extensions. Since MMX, Intel has released many vector extensions that increase the vector width, add floating point support, implement significantly more complex instruction operations, and even change the instruction format from two-address codes to three-address codes [5][6]. The major instruction set extension releases have always increased the vector width, with SSE implementing 128-bit vectors, AVX implementing 256-bit vectors, and AVX-512 implementing 512-bit vectors. Despite all the changes over the years, the naming conventions used throughout the designs have followed a consistent pattern.

The eight MMX vector registers are named $mmx0 - mmx7$ and are directly mapped onto the 80-bit floating point registers already implemented in the processor. The larger vector registers used in the newer vector extensions use their own designated register file. SSE implements sixteen 128-bit vector registers named $xmm0 - xmm15$ and AVX similarly implements sixteen 256-bit vector registers named $ymm0 - ymm15$. AVX-512 expands the number of vector registers to thirty-

two, adding registers $xmm16 - xmm31$, $ymm16 - ymm31$, and its own 512-bit registers named $zmm0 - zmm31$. These registers share the same register file, with the *xmm* registers simply using the lower 128-bits, the *ymm* registers using the lower 256-bits, and the *zmm* registers using the entire 512-bits for a given register file entry.

Several times throughout this thesis, we include vector assembly codes as examples. It can be quite challenging to read these codes as there is a lot of information encoded into every vector instruction's mnemonic. For the purposes of this thesis, the three most important pieces of information to look for are the name of the instruction operation itself, the data element size, and the vector width. The operation name is usually in the middle of the mnemonic and is equivalent to the scalar version of the operation; e.g., *paddd* performs **add**ition, *vandps* performs a logical **and**, *vmovapd* performs a data **mov**ement operation, etc. The data element size is encoded towards the end of the instruction mnemonic and is done according to Figure 2.1. Finally, the vector width can be inferred by the register names used in the instruction operands, which we previously enumerated.

Code	Size & Type
b	8-bit integer
w	16-bit integer
d	32-bit integer
q	64-bit integer
ps	32-bit float
pd	64-bit float

Figure 2.1: Vector Instruction Data Element Size Codes

2.2 Vector Loop Terminology

Loops are a fundamental concept in programming languages and have directly influenced many microarchitecture optimizations. This thesis seeks to optimize vector loops, but vector and scalar loops alike share the same terms to refer to their various components. Figure 2.2 gives a concise example of a scalar loop that introduces many of these terms.

Skipping the requisite variable initialization code, the variable *iter* is referred to as the loop **induction** variable. The induction variable is initialized at the beginning of the loop, increments or decrements each loop iteration, and often triggers the stop condition for the loop. The value by which the induction variable increments

```
int sum = 0, size = 1024;
int a[size], b[size * 4];
for (int iter = 0; iter < size; iter += 1) {
    sum += a[iter] * b[iter * 4];
}
```

Figure 2.2: Loop Terminology Example

or decrements each iteration is called the loop **stride**, and is 1 in this example. The minimum and maximum values of the induction variable throughout a loop's execution are collectively referred to as the loop **bounds**, which are $(0, size - 1)$ in this example.

Modern compilers often have the capability to automatically transform many scalar loops into equivalent vector loops [7]. The algorithms required to perform these transformations are computationally expensive and complex, but are done with the goal of making the compiled binary significantly faster at run-time. Auto-vectorization methods in compilers introduce a few more terms that are relevant for the loop in Figure 2.2.

While the memory accesses to array a will be adjacent for two successive loop iterations, the memory accesses to array b will be a constant width apart each iteration. When auto-vectorized by the compiler, the load to array a will be a normal vector load, but the load to array b will be called a **gather**. Similarly, memory stores that follow this same pattern are termed **scatter** operations, and are common in codes auto-vectorized by the compiler.

Finally, the scalar variable *sum* is called a **reduction** variable. The transformation required to vectorize this scalar update is more sophisticated than a scatter or gather, but is still very common in auto-vectorized codes.

2.3 Out-Of-Order Processing

Out-of-order processors allow instructions to be issued for execution as soon as all of their data dependencies are available. This is in direct contrast to in-order processors where instructions can only be issued in exact program order. An out-of-order processor implementation introduces many changes to an in-order processing pipeline [8], including additional pipeline stages and hardware structures.

Figure 2.3 shows the simplified structure of an out-of-order pipeline. The front

end fetches instructions to be executed and decodes them into their structural components, all in-order. The back end performs the issue, execution, and writeback of instructions out-of-order, before committing the instructions in-order. In hardware, the structures and logic for the commit stage are placed between the decode and issue stages so that it can coordinate the out-of-order execution with the rest of the pipeline. This coordination includes handling and sending squash signals received from the back end to the front end.

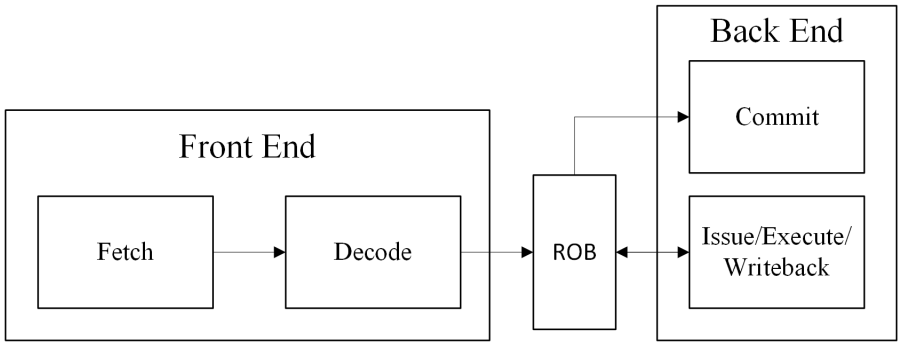

Figure 2.3: Out-of-order Processing Pipeline

The hardware structure that keeps track of in-flight out-of-order instructions and ensures instructions are committed in-order is called the Re-order Buffer (ROB). As long as an instruction is in the ROB and has not been committed, that instruction will not have persisted its data to the register file or to memory. As a result, the ROB enables the implementation of highly-aggressive speculative execution optimizations which would otherwise be impractical in in-order processors.

Chapter 3

Related Work

3.1 The ARM Scalable Vector Extension

In 2017, ARM announced a novel vector processing implementation called the Scalable Vector Extension (SVE) [9]. ARM previously implemented vector instructions tied to a specific vector width which was very similar to x86's implementation, but ARM took a different approach to supporting wider vector widths going forward. SVE makes vector instructions "vector-length agnostic" and allows processors to implement any vector width that is a scalar multiple of 128 bits, up to 2048 bits. By adding a new set of "while" instructions to the instruction set and utilizing extensive amounts of predication in the architecture, SVE allows the same vector code to run on multiple vector widths without recompilation.

While SVE is a promising solution to dynamic vector processing, it required extensive additions to the instruction set and vector hardware, and only works on programs compiled with SVE available. Due to our desire to allow legacy vector codes to be widened without recompilation and the immense complexity that would be required for an equivalent extension to x86, this thesis takes a different approach to dynamic vector processing.

3.2 Speculative Code Compaction

Moody et al. [10] showcased the potential for highly speculative binary transformations by dynamically compacting code present in a x86 processor's microop cache. Since x86 instructions are complex and the instruction set changes over time, the processor decodes these complex "macroop" instructions into a smaller set of more

simple instructions called "microops". Decoding macroops into microops is a high-latency process, so x86 processors implement a cache specifically for microops that allows the decode pipeline to minimize time spent decoding complex instructions.

Speculative Code Compaction increases the benefits of the microop cache even further by attempting to dynamically optimize instruction sequences in the microop cache, before storing the more efficient version in a designated partition of the microop cache. Most notably for this thesis, the analysis and transformation of these instruction sequences is asynchronous from the rest of the pipeline. This implementation benefits from minimizing any additional latency to the pipeline, but requires a highly complex implementation in the processor to ensure correct execution. In addition, the transformations performed in Speculative Code Compaction require that the first instruction in the optimized sequence is unchanged and serves as a prediction source for future speculative instructions. Section 4.2 details why this speculative execution design is unfit for widening vector instructions, and therefore disqualifies a simple extension to the Speculative Code Compaction framework for this use case. However, due to the efficiency of the asynchronous transformation, this thesis leaves the integration of the two optimizations for future work.

Chapter 4

Problem Space Exploration

Since dynamic binary vector widening is a novel concept, we need to break down the problem into its basic parts and carefully examine how the nuances of those parts affect a potential implementation. At its most basic level, vector widening takes as input a vector instruction of a given vector width and transforms it into an equivalent instruction that uses a multiple of that width. This transformation is only useful if the wider vector instruction is performing work requested by the program, and more importantly, does so safely without altering the logic of the original program. These two points form the minimum requirements of vector widening and will need to be kept in mind while we dive deeper into the more specific complexities of the process. To showcase these complexities, we examine what the complete vector widening process looks like on a simple code example.

4.1 Simple Example

Vector instructions can be utilized in any programming context, but we focus on vector loops as they have a variety of qualities that make them amicable to vector widening. Additionally, scalar loops in user code are often the target of optimizing compilers, which will try to automatically transform the scalar loops into equivalent vector loops.

Figure 4.1a shows a simple scalar loop that adds each integer array element $b[i]$ and $c[i]$ and stores the result in array element $a[i]$. This code is easily vectorized by the GCC compiler due to the loop bounds being known at compile time, the loop stride being constant, and all memory accesses within the loop being indexed by the loop induction variable. The auto-vectorized x86 assembly code is shown in Figure 4.1b, with the loop unrolled and the code annotated for readability. In line with

Figure 4.1: Simple Vector Loop Widening

(a) Scalar C Code

```
int a[256], b[256], c[256];
void example() {
  for (int i=0; i<256; i++){
    a[i] = b[i] + c[i];
  }
}
```

(b) Vectorized x86 Assembly

```
.Loop.Iteration1:
  movdqa b(%rax),%xmm0     // Load 128-bits from (b+%rax), store in %xmm0
  add    $16,%rax          // Increment %rax by 16
  paddd  c-16(%rax),%xmm0  // Load 128-bits from (c-16+%rax), add to %xmm0
  movaps %xmm0,a-16(%rax)  // Store %xmm0 to (a-16+%rax)
  cmp    $64,%rax          // Compare %rax to 64
  je    .Loop.End          // Jump to .Loop.End if %rax equals 64
.Loop.Iteration2:
  movdqa b(%rax),%xmm0
...
```

(c) Vector Widened x86 Assembly

```
.Loop.Iteration1:
  vmovdqa  b(%rax),%ymm0            // Load 256-bits ..., store in %ymm0
  add      $16,%rax
  vpaddd   c-16(%rax),%ymm0,%ymm0   // Load 256-bits ..., add to %ymm0
  vmovaps  %ymm0,a-16(%rax)         // Store %ymm0 to ...
  cmp      $64,%rax
  je      .Loop.End
.Loop.Iteration2:
  add      $16,%rax
  cmp      $64,%rax
  je      .Loop.End
.Loop.Iteration3
  vmovdqa  b(%rax),%ymm0
...
```

one of the listed motivations in Section 1.1, our code is compiled to a vector width of 128-bits but our processor supports up to 256-bit wide vectors. Therefore, we attempt to widen the vector instructions to 256 bits.

The code section ".Loop.Iteration1" in Figure 4.1c shows what this transformation looks like. We simply need to change each 128-bit vector instruction to its corresponding 256-bit vector instruction, as well as change the operands to use the 256-bit vector registers. The transformation of the 128-bit *paddd* instruction to the 256-bit *vpaddd* instruction is slightly more complicated since AVX arithmetic instructions use a three-address format, but this is still a straightforward mapping. Since the induction variable increments by the vector width each iteration and all vector memory accesses are solely based on the induction variable, we observe that the widened vector instructions are now performing two loop iterations worth of vector code in one iteration. This allows us to eliminate the vector instructions in the second loop iteration without any loss of functionality, as shown in the code section ".Loop.Iteration2" in Figure 4.1c. As a result, we have successfully widened vector instructions, such that we are making better use of the processor's vector hardware while eliminating instructions in the meantime.

4.2 Correct and Safe Execution

Unfortunately, this transformed code is only equivalent to the original code if a very important condition is met at run-time. In the assembly code in Figure 4.1c, there is a conditional branch instruction (je) between loop iterations one and two that must be *not taken* in order for the widened vector instructions to be valid. If the branch resolves to *taken* and we commit the widened vector instructions, we would have altered the original logic of the source program and the execution would not be correct from this point onward.

The obvious approach to handling this situation is to treat vector widening like more common speculative execution strategies. In branch prediction, a prediction is made regarding the direction of a conditional branch and any successive instructions are issued speculatively, allowing the processor to squash the speculative instructions from the pipeline if the prediction was incorrect when the branch is resolved. In vector widening, we need to predict that the conditional branch instruction will take the program to another loop iteration before speculatively issuing widened vector instructions. In the case that our prediction was incorrect, we need to squash the speculative instructions from the pipeline and resume execution from the point at which the prediction was made. An example for each of these speculative execution scenarios is shown in Figure 4.2.

```
// Misprediction must resume execution at .FarAway
je    .FarAway        // Prediction Source
movdqa b(%rax),%xmm0  // Speculatively Issued
add    $16,%rax       // Speculatively Issued
```

(a) Branch Prediction

```
// Misprediction must resume execution at vmovdqa
vmovdqa  b(%rax),%ymm0  // Speculatively Issued
add      $16,%rax       // Speculatively Issued
cmp      $64,%rax       // Speculatively Issued
je      .FarAway        // Prediction Source
```

(b) Vector Widening

Figure 4.2: Speculative Execution Sequence Comparison

As opposed to branch prediction, the speculative instruction sequence in vector widened code executes *before* the prediction source (the *je* instruction). This is a fundamental difference between speculative vector widening and other speculative execution scenarios, and therefore requires a different approach to implement safe speculative execution and recovery in the processor. Before moving on to our general algorithm for speculative vector widening and our solution to this specific problem, we examine a few complex vector widening scenarios and their challenges.

4.3 Complex Vector Widening

What made the example "simple" in Section 4.1 was that we did not have to add any new instructions in order to create an equivalent and wider vector code. If our target vector loop contains loop-invariant vector data or loop-carried dependencies, then we will need to introduce additional logic into the code when widening.

Figure 4.3 shows an example for both loop-invariant vector data and a loop-carried dependency. When the store to the array a with the constant 2 is vectorized by the compiler, a vector register will be filled with that constant value before the loop begins so that the loop only needs to issue a store from that vector register to memory. If we simply widen this store instruction, only half of the array elements would actually be set with the constant. Since the initial instruction that filled a vector register with that constant only did so for the original vector width, the

```
int a[256], b[256];
for (int i = 0; i < 256; i++) {
    a[i] = 2;
    b[i] = i;
}
```

Figure 4.3: Complex Vector Loop Widening

widened portion will not contain the constant and our widened store will not be correct. Therefore, we would have to issue a new instruction that fills the uninitialized portion of the wider vector register holding the constant value before speculatively issuing our widened code.

The loop-carried dependency seen with the store to the array b with the variable i is even more difficult to handle. When vectorized by the compiler, two vector registers will be allocated: $xmm0$ and $xmm1$. If each vector contains n data elements, $xmm0$ will contain the values $0, 1, ..., n-1$ and $xmm1$ will contain the value n at each element. With these two vector registers initialized, the loop will first issue a store from $xmm0$ to memory and then will increment $xmm0$ by $xmm1$. Vector register $xmm1$ is loop-invariant and can be widened with the method previously discussed, but $xmm0$ changes each iteration which is more difficult to widen. In order to initialize this register with correct values, we would have to store the addition of the original $xmm0$ and $xmm1$ registers in the uninitialized portion of the wider vector register.

While both of these complexities do not prevent using vector widening, issuing new instructions that are not in the original program significantly complicates a hardware implementation. As a result, this thesis focuses on widening the simple case and we leave implementing the more complex cases for future work.

Chapter 5

Architectural Overview

Armed with an understanding of the complexities of vector widening, we now detail the logic and hardware structures required to implement vector widening in the processor. We scope our design to only handle vector codes devoid of loop-invariant vector data and loop-carried vector dependencies to appropriately limit the complexity of the implementation. First, we describe our generalization for the vector widening process seen in the "simple" case in Section 4.1.

5.1 Vector Widening State Machine

So far, we have used an entire vector loop's source code to determine if it could be widened, but this is not a practical approach to take during program execution. Analyzing a variable length instruction stream to determine if it can be widened would have to run asynchronously from the rest of the pipeline and would require significant overhead to implement correctly. It is much more reasonable and efficient for us to fold vector widening as smoothly into the preexisting processing pipeline as possible. Therefore, our algorithms for detecting a vector loop, analysing it to ensure it can be safely widened, and transforming instructions to wider versions, only operate on one instruction at a time. We clarify this process by introducing the vector widening state machine, shown in Figure 5.1.

The state machine is defined by three distinct states: "Detection", "Analysis", and "Transformation". The machine is initialized to be in the "Detection" state and takes the instruction currently being decoded by the pipeline as input. The succeeding sections go into detail on what occurs at each state and what triggers the transitions between them.

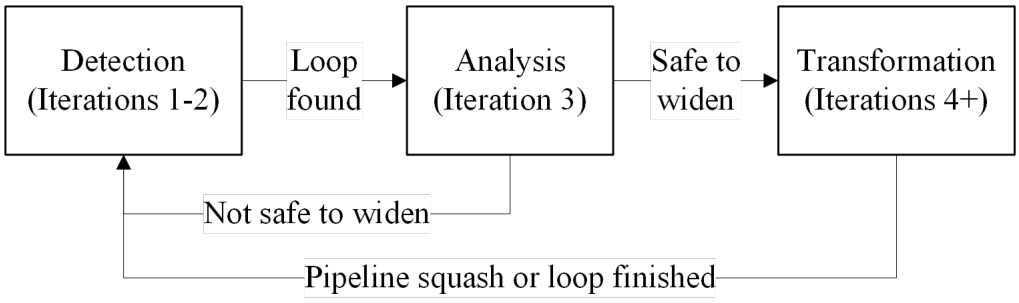

Figure 5.1: Vector Widening State Machine

5.1.1 Detection

Instructions themselves do not store information indicating that they are part of a loop, so it is necessary to implement a mechanism that will dynamically detect loops during run-time. This mechanism is usually already implemented in many modern high-performance processors, and is typically called a loop stream detector. The "Detection" state therefore does not need any significant logic to detect a loop, but rather observes the loop stream detector output and transitions to the "Analysis" state if a loop is detected. It is important to note that the loop stream detector cannot detect a loop before seeing two full loop iterations occur. By extension, a transition out of this state cannot occur until that second loop iteration has completed decoding. This is due to the fact that loops are defined as repeating sequences of instructions, and it takes two loop iterations to confirm that a sequence is repeating [11].

5.1.2 Analysis

Once a loop has been detected, we then need to determine if the loop has vector instructions and that they can be safely widened. The "Analysis" state checks the following:

- At least one instruction in the loop must be a vector instruction.
- Only the final instruction can affect control flow, and it must be conditional.
- All vector registers must be set by an instruction before they are read.
- Identify general purpose registers that are serving as induction variables and increment by the vector width each loop iteration. All vector memory accesses must be based solely on one of these induction variables.

If all of these conditions are satisfied, the state machine will transition to the "Transformation" state. Otherwise, the loop has been deemed unsafe or ineligible for vector widening and the machine will transition back to the "Detection" state. The implementation of each of these checks is described in Section 5.3.1, but they can all be performed with only a single pass over the loop's instructions.

5.1.3 Transformation

This state implements the physical process of transforming vector instructions just as we did in Section 4.1. For every pair of two loop iterations, vector instructions are widened in the first iteration and dropped from the pipeline in the second. If the loop stream detector determines that the current instruction is no longer part of the loop or a squash is triggered somewhere in the pipeline, the state machine aborts the transformation and transitions back to the "Detection" state.

As introduced in Section 4.2, this state also needs to implement speculative execution mechanisms to ensure the instruction transformation is correct and safe.

5.2 Speculative Execution

Commonly implemented speculative execution logic in modern high-performance processors are currently unable to handle the unique speculative execution instruction sequences created by vector widening. Since the prediction source for vector widened code is executed after the speculative code, the pipeline cannot commit completed speculative instructions until the prediction source has been executed and its result validated. However, successfully predicting that the vector loop will execute for a second iteration is not enough to ensure correct execution. Any instruction in the speculatively issued instruction sequence could trigger a squash in the pipeline and would reset the processor's instruction pointer to a new program counter. In this situation, speculatively transformed instructions that were issued before the squashed instruction will remain in the pipeline, but receiving a squash signal would cause the vector widening state machine to transition out of the "Transformation" state. As a result, the remaining instructions required of a complete vector widening transformation would not be issued and the program execution would be incorrect from this point onward.

A solution to both of these problems is to make every pair of loop iterations that were transformed by the vector widening state machine completely transactional at commit. The first instruction of the first transformed loop iteration will mark the start of an "unsafe" sequence, and every instruction until the completion of

the second loop iteration will be marked as unsafe. An unsafe instruction will be blocked from committing until the entire unsafe sequence executed without triggering a pipeline squash. If any unsafe instruction is squashed, the entire sequence will be squashed from the pipeline and execution will resume at the start of the unsafe instruction sequence. This method has the effect of rolling back a vector widening transformation that was unsuccessful, and will execute the loop using the original program code. Figure 5.2 shows how the vector widened code from Figure 4.1c would be marked as unsafe.

```
.Loop.Iteration1:
  vmovdqa  b(%rax),%ymm0            // Unsafe Sequence Start
  add      $16,%rax                 // Unsafe
  vpaddd   c-16(%rax),%ymm0,%ymm0   // Unsafe
  vmovaps  %ymm0,a-16(%rax)         // Unsafe
  cmp      $64,%rax                 // Unsafe
  je      .Loop.End                 // Unsafe
.Loop.Iteration2:
  add      $16,%rax                 // Unsafe
  cmp      $64,%rax                 // Unsafe
  je      .Loop.End                 // Unsafe Sequence End
.Loop.Iteration3
  vmovdqa  b(%rax),%ymm0            // Unsafe Sequence Start
...
```

Figure 5.2: Speculative Execution Solution

Conveniently, utilizing this transactional commit method removes the need for any prediction to be made regarding the direction of the conditional branch between the first and second transformed loop iterations. At the beginning of a new transformation sequence, we check the branch predictor to determine if the upcoming conditional branch will result in a second loop iteration occurring or not. If it will not, the transformation is abandoned and the original program code is executed. If it will, the speculatively transformed code is issued. In the case that the branch predictor was incorrect, the pipeline will trigger a squash at the branch and our new speculative recovery method will correctly rollback the instruction pointer to execute the original code.

This method has the drawback of limiting the size of the vector loops that can be widened since two full loop iterations will need to be blocked at commit before

the instructions can be safely committed. It may also slow down the pipeline to the point that vector widening is no longer improving performance of the code, but these trade offs are left to be analyzed in later sections. Nevertheless, this design enables speculative dynamic binary vector widening to execute safely and correctly.

5.3 Hardware Implementation

With all of the methods required by dynamic binary vector widening now defined, we are able to detail the additional hardware needed to implement these methods in a processor pipeline. This thesis's implementation is specific to out-of-order processing pipelines, but can be adapted to in-order machines.

5.3.1 Vector Widening Unit

The most substantial addition we make to the processing pipeline is called the Vector Widening Unit. This unit implements the vector widening state machine and is added to the end of the decode stage of the pipeline. The Vector Widening Unit takes the current instruction, associated loop stream detector output, and any squash signals received from the commit stage as its input. The unit either outputs an instruction to the pipeline back end or drops the instruction from the pipeline. If the vector widening state machine is in the "Detection" or "Analysis" states, the original instruction is simply forwarded along through the pipeline. If it is in the "Transformation" state, the first loop iteration's vector instructions are widened, the second loop iteration's vector instructions are dropped, and all other instructions are forwarded as is. If a squash signal is input, the state machine automatically transitions to "Detection".

The "Detection" state does nothing until the loop stream detector indicates that a loop is occurring, in which case it triggers the machine to transition to the "Analysis" state.

The "Analysis" state adds various Boolean flags to maintain state and bit-vectors to analyze instruction operands, all of which are isolated to the Vector Widening Unit. We break down each addition by the checks this state must perform, as detailed in Section 5.1.2:

- One Boolean flag marks if a vector instruction has been seen yet, which is set based on a simple logical "or" with each instruction's vector flag. The state machine aborts analysis if this flag is not set before the last instruction in the loop.

- Another Boolean flag marks if a conditional control flow instruction has been seen yet, and is again set with a logical "or" of each instruction's relevant conditional control signals. The state machine aborts analysis if this flag is set before the last instruction in the loop.

- Determining if vector registers are set before they are read is slightly more complicated. Two bit-vectors are added that have a bit corresponding to each vector register, one bit-vector for register reads and another for register writes. Each vector instruction sets both bit-vectors with its register operands, and aborts the analysis if a read bit-vector register is set but the write bit-vector register is not. This check occurs between the two bit-vector updates to handle instructions that read and write to the same register.

- Induction variables are identified by specifically looking for an "add immediate" instruction. Two more bit-vectors are required that have a bit corresponding to each scalar register, one to mark verified induction registers and the other to mark vector memory reference "base" registers. If an "add immediate" instruction is seen and the immediate value equals the vector width, the induction register bit-vector is set for this register. When a vector memory reference is seen, the relevant bit-vector is set for the "base" register. When the final instruction in the loop is seen, the Vector Widening Unit checks that all bits set in the memory reference bit-vector are set in the induction variable bit-vector and aborts analysis if any check fails.

The "Transformation" state uses the loop iteration count from the loop stream detector to determine whether to widen vector instructions or drop vector instructions. What is required to actually widen a vector instruction is highly dependent on the target microarchitecture, which we discuss in greater detail in Chapter 6. For our target microarchitecture, we simply have to double an immediate value in the instruction. Dropping a vector instruction from the pipeline is as simple as outputting a signal from the Vector Widening Unit that prevents the instruction from being added to the next stage's instruction queue.

Since all of this logic is inlined into the decode stage of the pipeline and is reasonably sophisticated, the decode stage will require an additional cycle to perform the additional work. This is a conservative estimate, but calculating the latency of an actual Vector Widening Unit implementation is out of scope for this work.

5.3.2 Speculative Execution Additions

The hardware required to implement safe speculative execution is much less substantial than that of the Vector Widening Unit, but is spread out across a larger portion of the pipeline. Most significantly, three new Boolean signals need to be associated with each instruction in the pipeline, which we store within each instruction's Re-order Buffer entry. We call them "unsafeSequenceStart", "unsafe", and "unsafeSequenceEnd". The Vector Widening Unit sets each of these signals if it is in the "Transformation" state. Every instruction is marked as "unsafe", the first instruction of the first loop iteration is the "unsafeSequenceStart", and the last instruction of the second loop iteration is the "unsafeSequenceEnd". When an "unsafe" instruction is ready to be committed, it is blocked from doing so until every "unsafe" instruction until the "unsafeSequenceEnd" is ready to commit. This is implemented with a simple pointer to entries in the Re-order Buffer, and advances through "unsafe" instructions if their flags indicate being completed without requiring a squash.

If the commit stage receives a squash signal for an "unsafe" instruction, all "unsafe" instructions until "unsafeSequenceBegin" are also squashed. In addition, the squash recovery instruction pointer is set to the program counter of the "unsafeSequenceBegin" instruction. This is also implemented with a pointer that advances through entries of the Re-order Buffer and setting squash signals in those entries accordingly.

Chapter 6

Methodology

This chapter describes the methods used to implement and evaluate dynamic binary vector widening in a complete microarchitectural simulator. Our basic needs for a simulator are that it implements the x86 instruction set architecture on an out-of-order processor and supports cycle-accurate simulations. The gem5 simulator meets all of these requirements and serves as an approachable software code base to which we can realistically add a dynamic binary vector widening implementation. However, gem5 was missing or incorrectly implemented a few key components needed for our implementation. Before discussing the work we did to bring the gem5 simulator up to speed, it is beneficial to describe our target microarchitecture to give that work background and context.

6.1 Baseline Architecture

One of our primary motivations for performing this thesis's work was to allow heterogeneous architectures that disable the largest vector widths due to instruction set architecture constraints to re-enable those vector widths. Intel's Alder Lake heterogeneous microarchitecture disables AVX-512 on its performance cores for exactly this reason [2]. Alder Lake implements an out-of-order execution pipeline, a loop stream detector, and a 256-bit vector width in all cores on the processor. As a result, Alder Lake makes for the perfect target microarchitecture for our gem5 implementation and successive performance analysis.

6.2 Gem5 Extensions

While gem5 has a robust and comprehensive x86 implementation, it was missing two key components needed to implement dynamic binary vector widening.

6.2.1 Loop Stream Detector

Most obviously, gem5 does not contain a loop stream detector implementation. The output from the loop stream detector is essential for vector widening and therefore needed to be implemented. We used the algorithms and structures detailed in Kobayashi's "Dynamic Characteristics of Loops" work [11], but with a few important differences. When the loop stream detector adds an instruction that is currently on the stack, we simply push the repeated instruction to the top of the stack and store metadata for the stack distance of the repeated instruction. This is more efficient in hardware than Kobayashi's implementation and the addition of metadata in the loop stream detector stack has multiple benefits. Adding a counter to signify the number of times an instruction has been repeated enables the loop stream detector to determine the current loop iteration and the instruction that begins a new loop iteration. These two pieces of information are essential for the "Transformation" logic of the vector widening state machine and do not add substantial overhead to the loop stream detector.

6.2.2 SIMD Microops

The significantly more challenging problem to solve involved how gem5 implements x86 SIMD macroops. In x86, user programs use complex macroop instructions that are then decoded during program execution into one or more microop instructions, which are the instructions that actually get executed by the processing back end. This design has the benefit of making x86 assembly concise, descriptive, and easily adaptable over time, but adds significant logic to the processing front end. Gem5 implements the decoding of SIMD macroops into microops that perform the instruction, but the microops themselves are not SIMD instructions. Figure 6.1 shows an example of how a 128-bit vector addition with a memory operand is decomposed into microops by gem5.

Ignoring much of the boilerplate code in this macroop definition, we can see that the 128-bit macroop is decoded into two separate 64-bit memory loads, followed by two separate 64-bit additions. This will correctly execute the logic of the macroop, but these scalar microops are impossible to widen without adding additional instructions to the pipeline. Issuing new instructions would eliminate any potential

```
def macroop PADDD_XMM_M {
    ldfp ufp1, seg, sib, "DISPLACEMENT", dataSize=8
    ldfp ufp2, seg, sib, "DISPLACEMENT + 8", dataSize=8
    maddi xmml, xmml, ufp1, size=4, ext=0
    maddi xmmh, xmmh, ufp2, size=4, ext=0
};
```

Figure 6.1: Gem5 SIMD Macroop Example

performance gain of vector widening as the original vector code and widened vector code would issue the same number of total instructions. As a result, we had to undertake the implementation of SIMD microops in gem5 and change all SIMD macroop definitions to use them. Substantial guidance and code snippets were applied from Zhengrong Wang's AVX-512 implementation in gem5 [12], but this implementation contained a few critical bugs and was missing many of the vector macroops and microops we required. Implementing the SIMD functionality needed for this thesis in gem5 was an immense software engineering challenge, which resulted in limiting the implementation for scope to integer arithmetic and data movement microops.

6.3 Limitations and Benchmarks

While the previous two extensions to gem5 are necessary in order to implement vector widening, there is one other missing component in gem5 that would have been out of scope of this thesis to implement.

Gem5 only implements the MMX, SSE, and SSE2 x86 vector extensions. This notably excludes all macroop definitions for all later vector extensions, but also the new instruction formats introduced by AVX and AVX-512. Both of these additions would require an expert familiarity with the complex x86 decode pipeline and an exceeding amount of time to implement all of this functionality correctly. As a result, gem5 cannot run x86 programs that use any vector extensions newer than SSE2.

Despite the immense amount of new functionality this thesis added to gem5, the many missing pieces greatly limit the types of vector programs that could potentially be widened by our implementation. We cannot widen any vector instructions that decode into unimplemented SIMD microops, any vector loops that have loop-invariant data or loop-carried vector dependencies, or decode any vector instructions

newer than SSE2. Side-effects of these limitations are that only 128-bit vector codes can be widened to 256-bits, and the performance of any widened vector codes cannot be compared with the 256-bit equivalent vector code.

After a detailed investigation of commonly used microarchitectural performance benchmarks like PARSEC and SPEC [13][14], it was clear that any real-world vector code would very unlikely be able to be widened by our implementation. Our response to this challenge is to utilize a microbenchmark suite within the GCC compiler infrastructure, called GCC Loops [15]. This suite contains twenty-one distinct and diverse scalar loop microbenchmarks that are able to be auto-vectorized by the GCC compiler. Each microbenchmark is intended to showcase a unique auto-vectorization capability of the GCC compiler and in turn generates twenty-one unique vector loops. We extend the suite to support gem5's checkpoint functionality, as well as hard-code the number of times each microbenchmark is run to 1024 to enable consistent and meaningful comparisons among the suite. Since auto-vectorization is one of the primary methods by which vector codes are created, GCC Loops serves as an effective metric for analysing the capabilities and performance of our vector widening implementation. We compiled the microbenchmark suite using GCC version 7.1.0 and with the following flags:

```
-O3 -msse -msse2 -ftree-loop-vectorize -ftree-slp-vectorize
```

Chapter 7

Results

The results of our dynamic binary vector widening implementation in gem5 are composed of two parts: 1) A performance analysis of the GCC Loops microbenchmarks and an examination of the reasons why many vector loops were not able to be widened, and 2) An estimate of the hardware overhead vector widening adds to an Alder Lake processor.

7.1 Performance and Capability Analysis

7.1.1 Sucessfully Widened Microbenchmarks

Figure 7.1 and Figure 7.2 show the instruction count improvement and execution time speedup for each of the vector loops, respectively. Microbenchmarks 1, 2b, and 4b saw a tremendous reduction in the instruction count, signaling that they were successfully widened by the Vector Widening Unit.

Microbenchmark 1 is the simplest of the three successfully widened vector loops and is actually the same code used in our simple vector loop example shown in Figure 4.1a. The fact that this code slowed down despite reducing the instruction count reveals a shortcoming of our speculative execution implementation. We explained the need for the conditional branch between two transformed loop iterations to be *taken* if an unsafe transformation is underway in Section 5.2. Our solution was to lookup the branch prediction for the end of the first transformed loop iteration at the beginning of that loop iteration, whereby the transformation is abandoned if the branch is predicted as *not taken*. Once the transformation begins however, that branch must be predicted as *taken* regardless of what the branch predictor actually outputs at that point, otherwise the unsafe sequence will not be completely issued

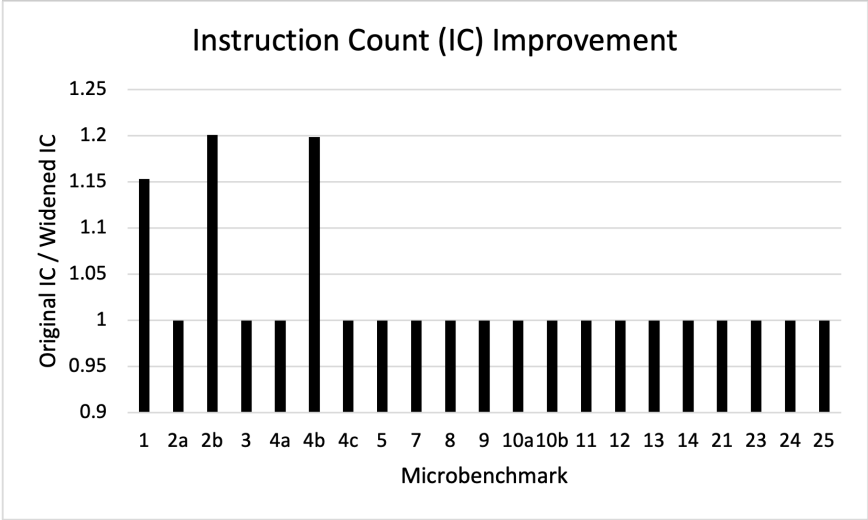

Figure 7.1

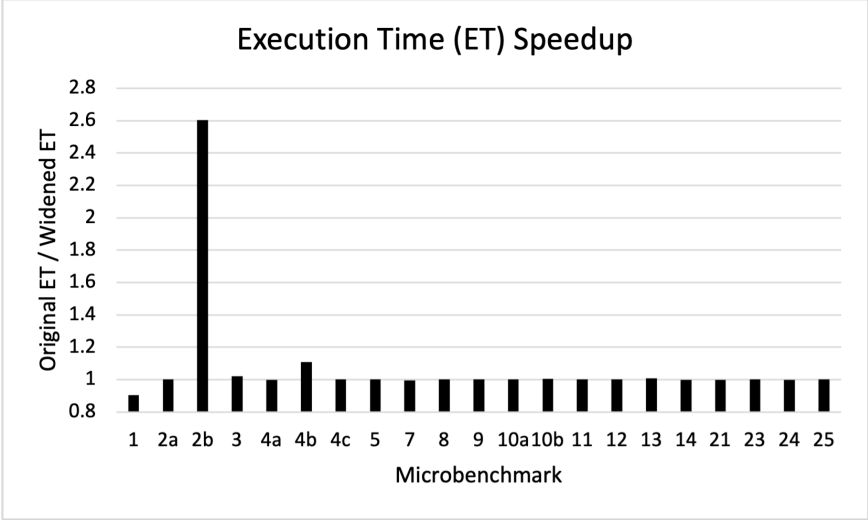

Figure 7.2

and the Re-order Buffer will not be able to commit or squash the unsafe sequence.

While this implementation ensures correct execution and does not cause slowdowns for simple branch predictors, it creates a significant source of inefficiency when a loop termination predictor is implemented in the processor. Since simple branch predictors can only predict short patterns of branch histories, a loop termination predictor is added to specifically track branch instructions that are *taken* many times before a single *not taken*. A loop termination predictor causes loops to become more efficient over time, but it relies on identifying exactly when that final *not taken* occurs in the branch history. Our manual overwrite of the branch prediction output for unsafe instruction sequences essentially prevents the loop termination predictor from ever identifying when a loop ends, but only if the loop ends in the middle of a unsafe instruction sequence. As a result, this vector loop will consistently issue too many loop iterations. These additional instructions do not affect the instruction count since they are squashed, but do cause a slowdown in the execution time of the program. Changing our speculative execution implementation to fix this source of inefficiency would cause this microbenchmark to speedup tremendously, but is a complex change and is thus left for future work.

Microbenchmark 2b performs very similar vector operations as microbenchmark 1, but the loop bounds are not known at compile-time. The resulting vector code therefore has to be more careful with not overflowing the loop bounds, and has to execute the final loop iterations using scalar code if the loop iteration count is not a multiple of the vector width. This code is not only able to be widened, but sees a tremendous 2.6x execution time speedup. This is due to the fact that the loop termination luckily does not occur in the middle of an unsafe sequence, and is able to benefit from the loop termination predictor's output. The performance of this microbenchmark reveals the potential of vector widening, and shows the performance improvement that could be seen by all widened codes with a less restrictive speculative execution implementation.

Microbenchmark 4b performs very similar vector operations as microbenchmark 1 and has the same loop definition, but the memory accesses to array b and c are not aligned along a memory boundary. Memory alignment occurs when the memory address holding a data element is a multiple of the size of that data. For example, a 2-byte data element would be aligned at memory address $0x0$ but unaligned at memory address $0x1$. Again, it is encouraging that this this code is able to be widened, but it suffers from the same inefficiency with the loop termination predictor as microbenchmark 1. This code sees a speedup however simply due to the fact that the vector loop iterates for 4x the number of iterations as microbenchmark 1. As a result, more instructions are skipped by widening than are squashed due to

the speculative execution design problem, causing an overall improvement of the program's execution time.

7.1.2 Un-widened Microbenchmarks

Despite the positive results that three distinct vector loops were able to be speculatively widened with our implementation, eighteen of the microbenchmarks were deemed ineligible for vector widening. The positive takeaway from Figure 7.2 is that the pipeline is not slowed down due to the addition of the Vector Widening Unit. Section 5.3.1 described the need to add an additional cycle to the decode stage when vector widening is enabled, but this additional cycle did not affect execution time for any of the microbenchmarks.

Microbenchmark	Vector Widening Result
1	**Successfully widened**
2a	Loop-invariant
2b	**Successfully widened**
3	Branch in Loop
4a	Loop-invariant, function call in loop
4b	**Successfully widened**
4c	Loop-invariant, unsupported vector instruction
5	Loop-invariant
7	Loop-invariant, branch in loop
8	Loop-invariant
9	Loop-invariant, loop-carried dependency
10a	Unable to validate vector memory accesses as strided
10b	Loop-invariant
11	Loop too large for widening
12	Loop-invariant
13	Loop-carried dependency
14	Loop-invariant
21	Loop-carried dependency, negative loop stride
23	Loop-invariant
24	Loop-invariant, unsupported vector instruction
25	Branch in loop, unsupported vector instruction

Figure 7.3: GCC-Loops Vector Widening: Transformation Results

Table 7.3 lists each of the reasons why a microbenchmark was not able to be widened. The most common disqualifying condition was the presence of a loop-invariant vector register, appearing in over half of the microbenchmarks. The presence of loop-carried vector data dependencies was much less common, preventing widening in only three microbenchmarks. This is unsurprising given that compilers struggle to auto-vectorize scalar loops with loop-carried dependencies [7], making this program characteristic unlikely to appear in auto-vectorized loops. Two of the most interesting disqualifying program conditions were seen in microbenchmarks 10a and 11.

Microbenchmark 10a performs multiple array accesses with the same loop induction variable, but the arrays are of differently sized types. When vectorized by the compiler, only one induction variable is used but the vector memory accesses perform an arithmetic operation to determine the base memory address. This microbenchmark reveals that our method to validate vector memory accesses as strided to the vector width is too simple to handle more complex memory address calculations. However, handling this case would add significant overhead to the Vector Widening Unit and increase its latency.

Section 5.2 explained how the size of vector loops that can be widened is inherently limited by the size of the Re-order Buffer, and this is exactly the reason why microbenchmark 11 was not able to be widened. Since two full loop iterations could not fit into the Re-order Buffer, the vector loop was disqualified for widening.

7.2 Hardware Overhead

A detailed explanation of the hardware necessary to implement vector widening was given in Section 5.3, but now we estimate the total overhead of implementing this hardware on an Alder Lake processing core. We restrict our calculations to the storage overhead introduced by vector widening since calculating the logical overhead is out of scope for this work.

The large majority of information used by the Vector Widening Unit is received as input from prior pipeline stages and the output of the loop stream detector, but there are a few bit-vectors that are needed to perform the work in the "Analysis" state. Alder Lake has sixteen vector registers and a 256-bit wide vector register file, with each 64-bit portion of a vector register being directly addressable by instruction operands. Therefore, the two bit-vectors used to check for loop-carried dependencies in vector registers will each be $16 * 256/64 = 64$ bits long. For the vector memory access checks, a bit-vector is needed that can reference every general purpose integer register. X86 implements sixteen such registers, making these two bit-vectors 16-bits

long each. The Vector Widening Unit needs two bits to determine if a conditional branch or vector instruction has been seen in a loop, as well as two more bits to keep track of its on state.

For the speculative execution additions, three bits need to be added to each entry in the Re-order buffer. Alder Lake's performance cores implement a 512-entry Re-order Buffer [16], resulting in 1532 additional bits of storage in the processor back end. The two pointers that index into the Re-order Buffer, one that is needed for the unsafe commit logic and the other for the speculative recovery logic, will each require $\log_2 512 = 9$ bits.

Summed up, vector widening adds a total of 1714 bits of storage overhead to each processing core.

Chapter 8

Future Work

This chapter briefly enumerates a few topics and research projects that would greatly improve the results of this thesis, or are worthy of further exploration.

1. Implement a speculative execution design that does not manually overwrite the branch predictor output, thereby allowing all widened vector loops to benefit from the loop termination predictor.

2. Extend the Vector Widening Unit to support loops that have a negative stride. This is would require more complicated memory access analysis and would cause the first transformed loop iteration to be skipped and the second to be widened.

3. Implement support for loop-invariant vector registers and loop-carried vector registers. Section 4.3 discusses the requirements of this work in great detail, but significantly more vector codes would be able to be widened as a result.

4. Utilize the Speculative Code Compaction framework to make the vector widening logic asynchronous from the pipeline and remove the need for an additional cycle at decode. The "Analysis" state would operate on instruction sequences in the unoptimized microop cache partition, and the "Transformation" state would insert widened instructions into the optimized partition.

5. Extend Speculative Code Compaction to compact two two-address SSE vector instructions into one three-address AVX instruction.

6. Complete the x86 vector support in gem5. This is a massive project that would require significant effort from several contributors, but would allow for

wider vector transformations and more accurate performance analysis. More importantly, this work would make x86 vector research more approachable and would likely yield exciting advances within the field.

Chapter 9

Conclusion

This thesis introduced a novel microarchitecture optimization that intended to speed up vector loops by dynamically and speculatively widening them to a larger vector width. We first performed a thorough exploration of the vector widening design space, before defining all of the algorithms and hardware structures required to realize this optimization in the processor. We then implemented this entire design in the gem5 microarchitectural simulator for the x86 instruction set architecture, along with the significant contributions of a complete loop stream detector and improvements to the x86 vector processing implementation. Our implementation successfully widened three of the GCC Loops microbenchmarks, with one seeing a 2.6x execution time speedup. In general, the additional vector widening logic requires minimal hardware overhead and did not add any significant execution time for the microbenchmarks that could not be widened.

While vector widening's design space and use cases are limited, this thesis made several original contributions to the field of microarchitecture. We proved that highly aggressive speculative optimization strategies are realizable in the processor with minimal overhead, and hope that this work will open the door to more ambitious and capable vector optimizations in the future.

Bibliography

[1] Paul Alcorn. *AMD sets all-time CPU market share record as Intel gains in desktop and notebook pcs*. Feb. 2022. URL: https://www.tomshardware.com/news/intel-amd-4q-2021-2022-market-share-desktop-notebook-server-x86.

[2] Ian Cutress and Andrei Frumusanu. *The Intel 12th Gen Core i9-12900K review: Hybrid performance brings hybrid complexity*. Nov. 2021. URL: https://www.anandtech.com/show/17047/the-intel-12th-gen-core-i912900k-review-hybrid-performance-brings-hybrid-complexity/2.

[3] Nathan Binkert et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[4] Alex Peleg, Sam Wilkie, and Uri Weiser. "Intel MMX for Multimedia PCs". In: *Commun. ACM* 40.1 (Jan. 1997), pp. 24–38. ISSN: 0001-0782. DOI: 10.1145/242857.242865. URL: https://doi.org/10.1145/242857.242865.

[5] Chris Lomont. "Introduction to intel advanced vector extensions". In: *Intel white paper* 23 (2011).

[6] James R Reinders. *Intel® AVX-512 instructions*. July 2013. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html.

[7] Saeed Maleki et al. "An evaluation of vectorizing compilers". In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 372–382.

[8] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[9] Nigel Stephens et al. "The ARM scalable vector extension". In: *IEEE micro* 37.2 (2017), pp. 26–39.

[10] Logan Moody et al. “Speculative Code Compaction: Eliminating Dead Code via Speculative Microcode Transformations”. In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 162–180.

[11] Makoto Kobayashi. “Dynamic characteristics of loops”. In: *IEEE Transactions on Computers* 33.02 (1984), pp. 125–132.

[12] Zhengrong Wang. *Bring AVX support to GEM5*. Dec. 2020. URL: `https://seanzw.github.io/posts/gem5-avx/`.

[13] Christian Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton University, Jan. 2011.

[14] John L Henning. “SPEC CPU2006 benchmark descriptions”. In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[15] Dorit Nuzman. *Auto-vectorization in GCC*. Oct. 2011. URL: `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[16] Dr. Ian Cutress and Andrei Frumusanu. *Intel Architecture day 2021: Alder Lake, Golden Cove, and gracemont detailed*. Aug. 2021. URL: `https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3`.