12-2019

Extending the Capabilities of Von Neumann with a Dataflow Sub-ISA

Martin Cowley
Western Michigan University

Extending the Capabilities of Von Neumann with a Dataflow Sub-ISA

by

Martin Cowley

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science
Computer Engineering
Western Michigan University
December 2019

Thesis Commitee:

Lina Sawalha, Ph.D., Chair
Janos Grantner, Ph.D.,
Bradley Bazuin, Ph.D.,

Acknowledgments

I would like to thank my advisor, Dr. Lina Sawalha, whose help and guidance has been invaluable throughout my studies. I would also like to thank my committee members, Dr. Janos Grantner and Dr. Bradley Bazuin for serving on my committee.

Also, I would like to thank the WMU Faculty Research and Create Activities Award (FRACAA) for supporting this work.

Finally, thanks to my friends and family for their support.

Martin Cowley

Extending the Capabilities of Von Neumann with a Dataflow Sub-ISA

Martin Cowley, M.S.E.

Western Michigan University, 2019

Instruction set architectures (ISAs) such as x86, ARM, and RISC-V follow the control flow model of computation, where a program is defined as a sequence of instructions. Early processors executed instructions one-by-one based on the control flow of a program. Dataflow is an alternative model of computation that uses the availability of data to drive instruction execution. Any instruction can be chosen for execution, independent of the instruction order, as long as the data is available for that instruction. While modern processors incorporate concepts of the dataflow model in the microarchitecture, the implementation of the ISA, the amount of instruction level parallelism is still limited. Explicit dataflow architectures bring the concept of dataflow execution into the ISA. This increases the amount of parallelism, but also introduces problems, such as control flow bottlenecks, inefficient data structures, and lack of speculative execution, that have prevented dataflow architectures from surpassing Von Neumann for all applications. Rather than chose one model or the other, this work extends the RISC-V ISA with a *dataflow sub-ISA*. A microarchitecture implementation of this ISA is capable of executing both types of instructions: Von Neumann and dataflow. This thesis introduces a dataflow sub-ISA and determines when it is best to use the dataflow subset, and when the standard instructions give better performance. The ideal situations for dataflow are sections of code with simple control structures and irregular memory accesses. A program with regular memory accesses and simple control structures gave a speedup of 9% over the Von Neumann version, and a similar irregular application was estimated to have a speedup of 1.2x.

Contents

List of Tables

List of Figures

1 Introduction

Due to power constraints, improving performance by increasing the frequency of CPUs is infeasible. Computer architects have instead turned towards using instruction level parallelism (ILP) to increase performance. ILP refers to how many instructions can be executed at once. Many hardware and software techniques exist to take advantage of the maximum amount of parallelism in a program. For example, loop unrolling, pipelined CPUs, multithreading, multicore processors, and very long instruction word (VLIW) architectures.

The instruction set architecture (ISA) of a processor defines how the software and hardware interact. The ISA defines the set of instructions, addressing modes, data types, memory models, and number of registers. The design of an ISA has an important impact on the hardware implementation, performance, and even ILP.

ISAs are typically categorized into two main categories: complex instruction set computer (CISC) or reduced instruction set computer (RISC). CISC instructions are complex, often performing multiple functions for each instructions. This is contrast to RISC instructions, which use simple instructions and hardware. x86 is an example of a CISC instruction set, while ARM is a RISC ISA. VLIW is another type of ISA which defines instructions in blocks that are executed in parallel to increase ILP.

Many ISAs in use today are based off of Von Neumann architecture, which share the following properties:

- Program counter (PC) for fetching instructions
- Register file to store instruction operands
- Shared memory for instructions and data

One of the key features of this architecture is that it follows the control flow model of computation. The PC points to the next instruction to be fetched, and is incremented each cycle to fetch a sequence of instructions. Each instruction reads its operands from the register file, sends the data to the ALU and/or memory, and writes the result back to the same register file.

There are a few difficulties when trying to adapt this model to parallel processing. First, it is inherently sequential–each instruction is fetched based on program order. The second problem is that the register file system makes it difficult to detect independent instructions and increases the complexity of the hardware. Both of these problems are potential roadblocks to extracting parallelism.

Numerous hardware techniques have been developed, such as pipelining, superscalar processors, multicore, etc., to allow hardware to take advantage of ILP. However, this raises an interesting question: can we increase ILP at the level of the program? Specifically, how much ILP can Von Neumann ISAs extract from a program, and are there other types of ISAs that can extract more parallelism?

VLIW ISAs use compiler techniques to group instructions together for parallel execution. This simplifies the hardware needed for parallel processing, but complicates the compiler and can limit ILP because it only uses static techniques.

The dataflow model represents a program in a graphical form. Each node is an instruction, and arcs between nodes represent data dependencies. The dataflow model ignores the ordering of instructions, and instead focuses on the availability of data to drive the execution of instructions. The dataflow representation of a program explicitly shows the dependencies between instructions.

The motivation for this model comes from the search for increasing ILP. This model does not suffer from the problems discussed above. It is nonsequential because the order of instructions is not specified, only the dependencies between instructions. It also replaces the register file representation with explicit dependencies, making it easier to know which instructions can be executed in parallel.

This work proposes to extend an existing ISA with a dataflow sub-ISA. Instead of imple-

menting a new, separate ISA for the dataflow instructions, a sub-ISA simplifies the compiler and the hardware by extending a unified ISA with both types of instructions. A microarchitecture implementation of our design would be able to execute both Von Neumann and dataflow programs. We expect that this will improve performance by being able to switch between the two modes using whichever one is more suitable to the code. We extended the RISC-V ISA, a research based ISA [1], created a compiler toolchain, and implemented a simulation model for our microarchitecture.

1.1 Contributions

The contribution of this thesis are listed below:

- Dataflow extension of the RISC-V ISA as a sub-ISA.
- Implementation of limited forms of dataflow speculation (Loop Predictor and Prefetcher)
- Addition of a hybrid dataflow/Von Neumann CPU model to the *gem5* simulator.
- Dataflow compilation toolchain (limited capacity) that can compile standard C/C++ programs.
- Analysis of the dataflow capabilities and bottlenecks

The rest of this thesis is organized as follows: Chapter 2 discusses relevant background and related work. Chapter 3 describes in detail the proposed dataflow sub-ISA. Chapter 4 details the microarchitecture implementation we used for our evaluation. Chapters 5 and 6 give our methodology and results, respectively. Finally, chapter 6 concludes the thesis and discusses future work.

2 Background and Related Work

ILP can be quantified as the number of instructions that can be executed at the same time determined by data dependencies between instructions. Two instructions are independent if there are no data dependencies between them and can be executed in parallel or in any order. If the second instruction depends on the first instruction, then the second instruction must execute after the first one. Figure 2.1 shows two code segments, the first where each instruction is independent, and the second where each instruction is part of a chain of dependencies. Each instruction in the first program can be executed in parallel. Additionally, the order of instruction execution does not matter–independent instructions can be executed in any order. The second program is inherently sequential because each instruction must wait for its parent instruction to complete. Performance of a program is limited by the amount of parallelism intrinsic to a program and the amount of ILP that the hardware can extract.

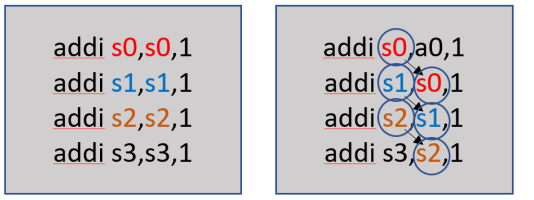

Figure 2.1: Difference between dependent and independent instructions

There are many ways to extract ILP. Pipelined processors execute multiple instructions simultaneously but in different stages. Superscalar pipelines fetch, decode, and execute multiple instructions simultaneously. Out-of-order processors dynamically reorder instructions to increase ILP. However, these are hardware implementations of a control flow ISA. This thesis examines the possibility of using the dataflow model of computation to improve performance,

which has the potential to exploit more ILP.

Processors with low levels of parallelism are susceptible to instructions with long latency. One example is a memory instruction that results in a cache miss. A simple sequential processor that must execute each instruction one at a time is susceptible to these instructions because it cannot move onto the next instruction until the previous one is finished. A cache miss that takes many cycles to resolve wastes time because there could be other independent instructions ready to be executed. Researchers turned towards the dataflow model to help alleviate the problem.

2.1 Model of Computation

A model of computation defines the way a program is executed. Most programming languages and processors adopt the control flow model, where a program is treated as a sequence of instructions. Conceptually, a control flow processor fetches and executes instructions in the order they are found in memory (called in-program order). This model is very simple and widely used in computation.

There exists another model of computation, dataflow, which takes a different perspective. Instructions are executed based on the availability of data, ignoring program order. This model is nonsequential and is able to execute any instruction that is ready. Researchers have been interested in the dataflow model to see if it can help CPUs exploit ILP in a program by focusing on data rather than program order, which is often incidental to the execution of the program.

An example of a dataflow graph is shown in Figure 2.2. Each node in the graph is an instruction and arcs between nodes represent true data dependencies. Nodes that are next to each other have no dependencies and can either be executed in parallel or in any order the processor decides. This makes parallelism between instructions explicit and can improve performance over sequential execution.

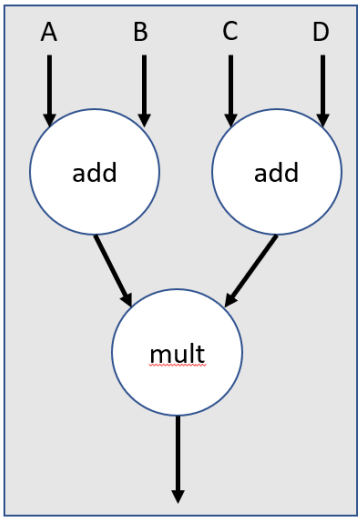

Figure 2.2: Dataflow Graph of formula (A+B) * (C+D)

2.2 Von Neumann Architectures

Von Neumann ISAs are sequential: a basic program is a sequence of instructions and the CPU follows the flow of control specified by the program. A program counter (PC) is used to keep track of the next instruction to be executed. It is either incremented after each instruction or, when a control instruction is reached, the PC is modified to begin executing a different part of the program. However, modern microarchitectures deviate from this pattern internally because of the limitations of sequential execution. Extra hardware is added to allow for dataflow execution.

With a Von Neumann program the dataflow graph above can be constructed by analyzing register access patterns of each instruction. Modern processors incorporate this concept into the hardware. Instead of executing sequentially, extra hardware is added to dynamically analyze dependencies between registers and execute instructions out of their program order to increase ILP. This technique is called out-of-order (OoO) processing and greatly improves performance. OoO processors can hide the latency of some instructions by executing other independent instructions instead of stalling the pipeline.

One limitation of this method is that these processors are still sequential at the ISA level. OoO processors fetch in-order and write back the result in-order to maintain consistency with sequential processing and support precise interrupts. This is done by loading instructions

into a re-order buffer (ROB), which acts like a "window of instructions" that allows dataflow execution while still maintaining program order. Any instruction in the ROB can be executed if it is ready and the processor will skip over instructions that are still waiting or are have not finished executing.

The size of the ROB limits ILP, because there could be ready instructions that have not yet entered the ROB. One solution is to increase the window of instructions. There is another type of architecture, explicit dataflow, that exposes the dataflow graph to the ISA level and increases ILP.

2.3 Dataflow Architectures

Explicit dataflow architectures differ because they expose the concept of a dataflow graph to the ISA level and execute the dataflow graph directly. Pointers to dependent instructions are encoded in the machine binary, so that each instruction explicitly points to its dependencies. Rather than fetching instructions using a PC, dataflow architectures use the dependency arcs in the dataflow graph to fetch new instructions, which could be anywhere in memory. This increases the window of instructions to the entire program and increases the potential amount of parallelism that can be extracted. When an instruction executes, the explicit pointers are used to fetch and activate dependent instructions rather than a program counter. Instead of a register file, data is sent directly to the required instructions. Many architectures have been developed for executing dataflow graphs.

The table at the end of the section gives an overview of different dataflow architectures. We categorize them with the following features: scheduling methods, code coverage, pure dataflow or hybrid architecture, programming language, and how the CPU was implemented. Other work related to dataflow architecture not included in the table are [2, 3, 4, 5, 6, 7, 8].

Direct Communication Conceptually, dataflow programs pass data from one instruction to the next. There are different ways to implement this in hardware. The first concept of an explicit dataflow architecture grew from the register file system used in Von Neumann architectures. The computer would contain a set of instruction cells (or registers). Each cell

contains space for the instruction and two operands. When the two operands arrive, the instruction and data are sent together to the function units (FUs) for execution. The result of the instruction is then sent directly to dependent instruction cells. Hence, we call this method direct communication. We chose not to use this method for our microarchitecture implementation because the number of instruction cells would severely limit the size of the code. The SEED [9] architecture utilizes direct communication, but only by limiting the scope of execution (discussed below). The TRIPS [10] architecture uses direct communication, but on the level of hyperblocks rather than individual instructions.

Tag Matching Tagged token architectures create a common pool of data to store operands until the corresponding instruction is ready. Each piece of data is stored in a token, which contains the data and an identification tag. An instruction becomes ready when both operands have arrived to the pool. This event is detected when two tokens with the same tag have reached the pool.

One common technique for dataflow execution is called tag matching. It uses random access memory (RAM) to temporarily store operands until their corresponding instructions are ready. This approach is used because it can store a large number of operands for many different instructions. The Manchester [11] machine and MIT [12] tagged-dataflow both independently took a step in this direction by using associative memory (full-associate memory or hash functions). The instruction address is used as a key to index the memory. This is done so that two operands destined for the same instruction will be mapped to the same location in cache. When this happens, the instruction is determined to be ready for execution and is sent to a FU along with the operands. We use the same process for our microarchitecture implementation using a fully associate cache to store operands. The Monsoon architecture [13] uses similar tag-matching hardware, but its goal was to replace the associate memory, which can be expensive, with a simpler approach [13]. A simple effective address calculation is used to determine where the token is stored in the pool. A program is statically split into frames, and each instruction is located at an offset from the current frame. A static instruction offset is added to a dynamic frame pointer to calculate the location in memory the token needs to go. This

introduces the problem of having to maintain the allocation and deallocation of each frame as part of the calling convention, but greatly simplifies the matching mechanism.

Programming Paradigms The programming methodology used for dataflow architectures has changed over the years. Early systems exclusively used functional or single-assignment languages. Examples languages include FORTRAN [14, 15, 16], SISAL [11], and ID [12, 13, 16, 17]. Single-assignment languages have variables that can only be assigned once. This greatly simplifies the compilation process for dataflow, but destroys legacy code written in imperative languages like C/C++.

More recent dataflow architectures recognize that they need to be programmed using standard languages. SEED [9], Wavescalar [18], and our design use benchmarks that are written in C++ so we can compile and run standard benchmarks written in imperative languages.

Memory Disambiguation Memory instructions can have hidden dependencies that are not revealed by analyzing register access patterns, because the addresses are not always known statically. Memory disambiguation is a class of techniques for overcoming this problem.

Early dataflow architectures did not use normal memory instructions. Manchester avoided the problem of memory disambiguation by not using memory instructions at all: all data is stored inside the processor, without any data memory access. They recognized that this was a serious bottleneck, since any program would run out of space when large data structures are used [11].

Many architectures solved this problem using a special type of data structure called I-Structures [19, 13, 17]. This extended the single-assignment language approach to memory instructions. Each location in memory can only be written to once. If two store instructions attempt to store at the same location, a runtime error occurs. If a load instruction tries to execute before data has been written, then it is added to a queue and waits. When the corresponding store executes, the data is sent to every value in the queue.

This solves the problem of memory disambiguation, but only by restricting the programmer to single-assignment paradigm. We chose not to use these data structures for the same reason we abandoned single-assignment languages in general. They are not compatible with

the C/C++ language and therefore not suitable for a hybrid architecture that can run realistic benchmarks.

We take a conservative approach that is also utilized by other architectures like SEED architecture [9]. We start by assuming that all memory instructions are dependent and must be executed in a sequence. We enforce this sequence by adding true data dependencies between each store-load instruction. This creates a chain of memory instructions; each memory instruction must wait for the previous one to execute. However, because this conservative approach forces instructions to be executed in a sequence without speculation, it can lower performance unnecessarily. We offset this problem by using a compiler optimization that removes memory instructions from the chain if it is known they are independent at compile time.

WaveScalar [18] removes the need for I-structures by having "wave-ordered memory" [18]. WaveScalar was one of the first dataflow ISAs to focus on the execution of conventional programs. They use a hardware implementation to chain dependent memory operations and force them to execute in sequence. This is similar to SEED but implemented through hardware structures instead of on the dataflow graph level.

2.4 Hybrid Model

The idea of hybrid dataflow architectures were discussed from the early days of dataflow architecture [17]. Iannucci discussed how the features of dataflow and Von Neumann could be combined to improve performance [17]. However, there are different ways of approaching a hybrid architecture and different definitions of the word hybrid.

Blending Iannucci pointed out that the dataflow architecture and the Von Neumann architecture were not orthogonal–instead, they lie on opposite sides of a spectrum [17]. He took features of dataflow architecture and added them to a Von Neumann base. Because OoO cores borrow dataflow properties, they can be considered a form of a hybrid Von Neumann/dataflow architecture. This is orthogonal to our microarchitecture, because we are not attempting to blend the properties of both architectures. Our work separates the two models into two distinct modes of operation.

The epsilon 2 architecture was an early a hybrid approach–a local PC was used for scheduling within a block, but chose which block to execute in a dataflow manner [16]. When constructing a dataflow graph, each node was a block of sequential instructions.

Another example is the TRIPS architecture, an implementation of the EDGE ISA. EDGE splits a program into blocks of instructions, called hyperblocks. In the TRIPS architecture, every instruction in a block is sent to a functional unit together. Data is passed between hyperblocks using dataflow scheduling, but execute in the Von Neumann way inside a block. This removes the global register file found in most Von Neumann designs, but each functional unit has a local register file.

Another notable work was Tartan, which used a reconfigurable fabric to remove the fetch-decode system found in Von Neuamann architecture. The fabric is made up of functional units that pass their data directly to where it is needed, without writing to a register file. The fabric is connected to a standard processor, making this a hybrid architecture. While this is a dataflow machine, their work is orthogonal to ours.

Mode The closest paper to our work is the SEED architecture [9]. They investigated the performance of a hybrid model with fine-grain switching. Our work is similar because we have two distinct modes of operation: dataflow and Von Neumann. The hybrid processor is able to switch back and forth between these two modes over the lifetime of a program. Our work is different than SEED because we implemented a sub-ISA and used a tagged-token pipeline for our dataflow architecture. Additionally, we implemented a simulation model to test our design, while SEED was evaluated using a high-level analysis.

Table 2.1: Overview of Dataflow Architectures

- Communication – how is data sent from one instruction to the next.
- Region – "full" indicates entire benchmarks can run on this architecture while partial means the dataflow hardware only supports a subset of code.
- Hybrid – "*" indicates this is a hybrid architecture
- Language – supported programming languages
- Implementation – the implementation method (simulation, RTL, or VLSI)

Name	**Communication**	**Region**	**Hybrid**	**Language**	**Implementation**
Dennis [14]	Direct	full		NA	Theoretical
Manchester [11]	TTDA	full		ID	VLSI
MIT [12]	TTDA	full		ID	VLSI
Monsoon [13]	TTDA	full		ID	VLSI
Wavescalar [18]	TTDA	full		C/C++	RTL
TRIPS [10]	Direct	full	*	C/C++	VLSI
Tartan [20]	Direct	full	*	C/C++	C Simulation
SPEED [9]	Direct	nested loops	*	C/C++	Simulation
This work	TTDA	full	*	C/C++	Simulation

3 RISC-V Sub-ISA

The design of an ISA has more to do than just selecting a set of instructions. The ISA of a processor does not specify all of the microarchitectural details, but it does define the basic modes of operation; it defines what the CPU is doing from the point of view of the program. For example, Von Neumann ISAs follows the standard register file, PC, and control flow model of computation.

The ISA is an important first step when designing a CPU. There are many layers of abstraction that are used in computer architecture. An ISA is the layer that defines how the software interacts with the hardware. There are many existing ISAs, for example x86, ARM, RISC-V, MIPS, Alpha, etc. The next question is why is it necessary to create special dataflow instructions. After all, the out-of-order model shows that many complex operations can be happening underneath the surface that the ISA does not know about. This simplifies the compilation process greatly, because it allows for limited dataflow without modifying the program. However, the existing out-of-order hardware demonstrates the difficulty with this approach. Dynamically analyzing the dependencies between instructions is difficult and expensive in terms of hardware and energy. We need a new ISA with instructions that could contain the dataflow graph explicitly. This greatly simplifies the microarchitecture since it does not require extra hardware for analyzing dependencies. However, it also complicates the compilation process because the dataflow graph needs to be statically constructed. A sub-ISA is advantageous because the microarchitecture can execute normal instructions as well as our dataflow instructions. A sub-iSA is slightly different than a normal hardware accelerator or coprocessor, which are more like peripherals where some of the work is offloaded from the host CPU. By implementing our sub-ISA, we are creating a single processor capable of executing dataflow and

control flow software.

3.1 RISC-V Base ISA

RISC-V is an ISA specifically designed for research. It is open source, meaning it can be used and modified by anyone in academia or industry. It was also designed to be customizable through different extensions. The base ISA is relatively small and only contains 32-bit integer instructions. It does not support multiplication and division. If more functionality is needed, extra extensions can be added on top of the base ISA. The existing list of extensions includes RV64I (64 bit instruction subset), RV32M (32-bit multiplication instructions), RV64M (64-bit multiplication and division instructions to RV32M), and RV32F/RV64F (includes floating point instructions). The base is required for all RISC-V implementations, but each of the extensions are optional. This allows each implementation to be customizable and small, using only the extensions that are needed. These features made the RISC-V ISA desirable for our research.

3.2 Dataflow sub-ISA

Creating an extension in RISC-V is a simple task. The designers left sections of the opcode space empty so new instructions can be inserted. Table 3.1 lists each instruction in the dataflow extension. It also shows the different fields for each instruction. Because dataflow instructions are radically different from their control flow counterpart, we can not reuse any of the instructions from the base ISA. We added dataflow versions of add, addi, sub, branch, multiply, etc. The next question is how does the hardware know to execute the instructions through a dataflow or von Neumann process? We created two different modes of operation, and added a control register that tells the microarchitecture how to behave. This is the only register added to the ISA, as the dataflow model does not read or write to registers. On startup, the architecture begins executing in control flow mode. In continues until it reaches a special switch instruction. The CPU will immediate switch to dataflow mode, and begin executing instruc-

tions in the dataflow manner described above. Communication is handled one of two ways. First, because the two models share memory, loading and storing data directly to memory is a simple way to communicate. This is used when large data structures are needed. The second way is through the use of special instructions that pass data between the two models (i.e. translate a register value into a data token, or write a token back into a register). The "tok" instruction is used by the Von Neumann pipeline to generate a data token from the value in a specified register.

Despite these theoretical advantages, dataflow architectures have yet to surpass state-of-the-art OoO processors. Some of the reasons include lack of compiler support, the difficulty of implementing speculation in dataflow architectures, and the fact that OoO cores are very good when running regular applications. Instead of trying to solve all of these problems and surpass OoO for all applications, we have extended the RISC-V instruction with a subset of dataflow instructions. The reason being that an architecture built from this ISA would be capable of executing both Von Neumann and dataflow programs with reduced overhead.

It is possible to improve performance by creating a hybrid integrated with multiple types of CPU models. This technique is called specialization; it comes from the recognition that one architecture cannot optimally handle all types of program behavior. Having specialized subsets would let us target different architectures for different application requirements.

3.3 RISC-V Dataflow Extension

Our ISA defines two modes of operation. In the standard mode, instructions are fetched and executed using the standard ISA. In the dataflow mode, instructions are executed as part of a dataflow graph. A switch instruction was added to allow for easy switching between the OoO execution and the explicit dataflow modes. Switching modes changes the driving force behind how instructions are fetched: the PC for the OoO and explicit pointers for the dataflow. Each destination also has a left/right bit, which distinguishes between the left and right operand for the instruction. This is important because some of the instructions, such as subtraction, require the ability to distinguish between left and right operands.

Dataflow Instruction Format Dataflow instructions take one of the forms in Figure 3.1. The second form replaces two of the destination operands with an immediate operand.

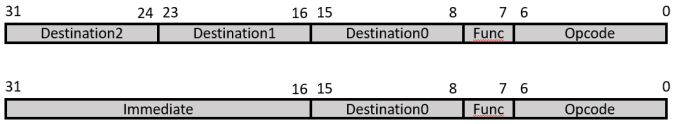

Figure 3.1: Dataflow Instruction Formats

Fan-out One of the efficiency of Von Neumann architecture is that once an instruction writes to a register, the result can be read from the register by multiple dependent instructions. With dataflow, however, the data must be sent to each dependent instruction separately. This is a problem because each destination address needs to be encoded directly into the machine code. Each instruction is 32 bits, which limits the number of destinations we can encode.

Fan-out describes the number of destination instructions. Many previous dataflow ISAs use a fan-out of two. However, we found that this produces serious performance bottlenecks.

Figure 3.2 shows the difference between a fan-out of two and three. A smaller fan-out limits the number of dependencies. In order for data to be sent to an arbitrary number of destination instructions, we use copy instructions. These instructions do not perform any operation; they simply pass on the data to more dependent instructions. By chaining a sequence of copy instructions, we can create as many copies as needed–but this affects performance.

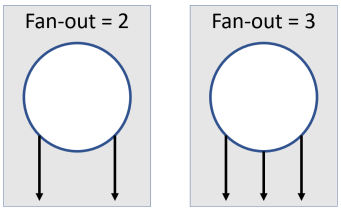

Figure 3.2: Fan-out defines the max number of output arcs.

This makes the fan-out size an important factor. Figure 3.3 gives an example of an instruction with 5 output arcs. If each instruction has a fan-out of two, three copy instructions need to be inserted into the program. Each copy forms a dependency chain, so they cannot be executed in parallel. Each one has to wait for the previous copy to execute.

There is a trade off between the fan-out and the size of the machine instruction. We can

not have an infinite number of destination, because each destination has to be encoded into the instruction binary. A fan-out of three reduces the number of copy instructions and is a reasonable number for an instruction size of 32-bits.

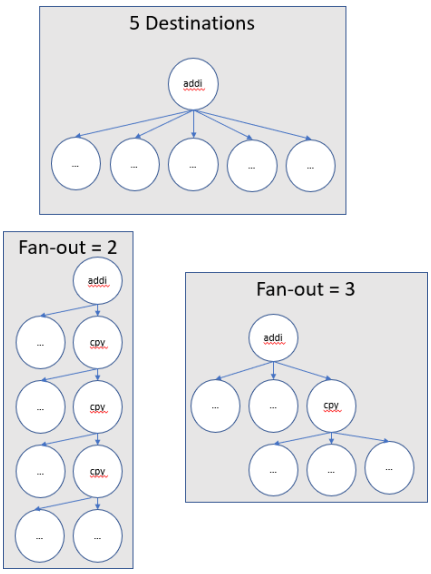

Figure 3.3: Increasing the fan-out

Monadic vs Dyadic Instructions For simplicity, we limit the number of operands for each instruction to either one or two. Monadic instructions require only one operand to fire. Dyadic instructions have two operands. Having instructions with more than three inputs would complicate the microarchitecture (described in the next section).

Dataflow Immediate Format While there are three slots for dependent addresses, two of the slots can be replaced with an immediate field. This transforms an instruction into a Monadic instruction, but lowers the fan-out to one and increases the number of copy instructions.

Dataflow Control Instructions An important question in any dataflow architecture is how to handle conditional execution, Specifically branch instructions. Branch instructions use conditionals to control the flow of the program. In Von Neumann architecture, this is done by modifying the PC. The dataflow counterpart works by having conditional execution paths. Standard dataflow instructions have multiple output arcs. Dataflow has two conditional output

arcs as shown in Figure 3.4. If the condition is true, then the input data will be sent along the left path. When the condition is false, the right arc is chosen.

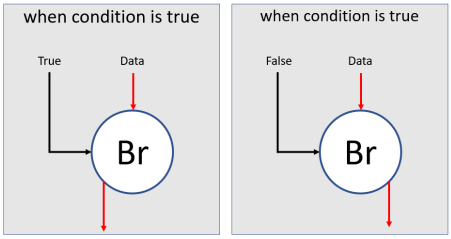

Figure 3.4: Branch instructions send data down either one path or the other depending on a boolean.

Instruction List The following table shows all of the instructions in our dataflow extension.

Table 3.1: Dataflow Extension

Operation	Addressing Mode	Description
df_add	dataflow	signed addition (+)
df_addi	df_immediate	Addition with immediate operand
df_call	dataflow	Function Call
df_ret	dataflow	Return from function
df_sl	df_immediate	shift left by immediate value
df_mult	dataflow	Signed multiplication (*)
df_sub	dataflow	Subtraction (-)
df_cmpLT	dataflow	Compare less than (<)
df_cmpEq	dataflow	Compare equal to (==)
df_cmpGT	dataflow	Compare greater than (>)
df_and	dataflow	bit-wise AND
df_not	dataflow	bit-wise NOT
df_cpy	dataflow	cpy input to all destinations
df_lw	dataflow	load word from memory
df_lwi	dataflow	load word from memory with immediate offset
df_br	control	branch between two possible destination arcs
df_switch	control	switch modes of operation
df_tok	control	create a data token using a value from a register

4 Dataflow Microarchitecture

This chapter describes the different stages and features of our dataflow implementation. We start with a discussion of each stage of the pipeline and how they differ from similar stages in Von Neumann architectures. The chapter also discusses the different features of the pipeline and how the dataflow ISA affects the structure of the microarchitecture. Finally it mentions the advantages and the bottlenecks of dataflow microarchitecture compared to Von Neumann microarchitectures.

4.1 Data Tokens

Conceptually, dataflow instructions send data to their dependent instructions when executed. However, typically the microarchitecture does not look anything like this. In hardware, data is represented in the form of data tokens, shown in Figure 4.1. When an instruction executes, one or more result tokens are generated. The address of the destination instruction is added to the token so the processor knows where to send the data.

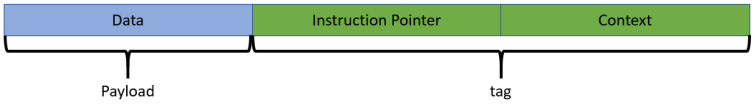

Figure 4.1: Data token

4.2 Dataflow Pipeline

The width of the pipeline is 8, and can execute up to 8 instructions every cycle. This parallels the superscalar architectures found in Von Neumann models. A simplified view of the dataflow

pipeline is shown in Figure 4.2. The decode, match, and execute stages each takes a minimum of one cycle. The fetch stage takes longer because the memory model we are using requires 3 cycles to access the first level cache. So each instruction takes a minimum of 6 cycles to be processed by the pipeline. One exception is the copy instruction, which is executed in the match unit. Copy instructions do not need a functional unit to execute, so they are not sent to the execute stage. This optimization saves a couple of cycles for each copy instruction.

Flow of data between stages Figure 4.2 shows how the different stages interact. When an instruction executes, the execute stage sends the resulting tokens back to the match unit. This is done to check if the instruction has been fetched already. If it has, then the instruction continues at this stage of the pipeline. If not, then the token is sent back to the token queue at the beginning of the pipeline.

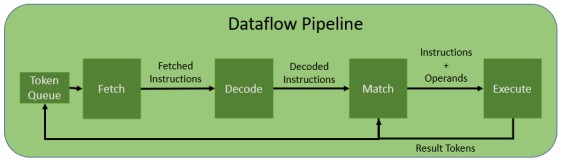

Figure 4.2: Pipeline diagram showing the connection between pipeline stages

Fetch and Decode The driving force of instruction execution is the token queue, which is a FIFO buffer which stores all of the data that needs to be processed next. Each cycle, the front tokens of the queue are popped. As mentioned in a previous section, each token contains a pointer to an instruction. The fetch unit sends this address to the cache port to fetch the corresponding instruction. Once fetched, the instruction and the token are sent together to the decode stage.

While the width of the pipeline is four, it is inefficient to restrict the fetch stage to this limit. The rest of the pipeline processes instructions faster than the fetch unit can keep up with. This severely limits performance, and causes sections of the pipeline to remain idle for significant periods of time. To alleviate this problem, we increased the fetch width to 12 and added a buffer between the fetch and decode stages. Up to 12 instructions can be fetched in each cycle. Everything that cannot be immediately processed by the decode stage waits in the buffer. This

helps smooth out the gaps in the pipeline execution and keep the pipeline full.

The decode stage does not differ significantly from its Von Neumann counterpart. It extracts the required information from the instruction. The biggest difference is that there is no register file to read from. The operands instead come from the data tokens. The following information is determined at this stage: destination addresses, the number of operands, immediate values, etc.

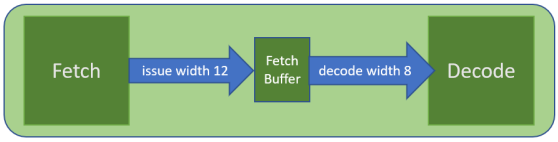

Figure 4.3: Fetch width is increased to hide fetch latency

Next, we handled another problem that was affecting performance: token shortage. Some programs do not have enough instruction level parallelism to keep the pipeline full. Sometimes the data is not being generated fast enough, so there are not enough tokens in the token queue to fetch instructions every cycle. This can happen with code that has a lot of data dependencies between instructions and not enough instruction level parallelism.

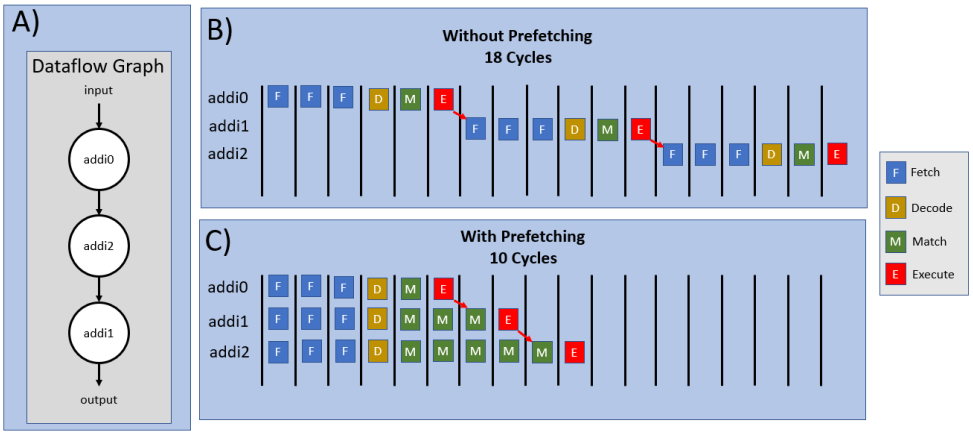

Figure 4.4: Dataflow graph and pipeline diagrams with and without prefetching

An example is given in Figure 4.4. Figure 4.4.A is a dataflow graph that shows a small section of a program with three *addi* instructions: addi0, addi1, and addi2. The important feature of this code is that the instructions must be executed in a sequence because there is a data dependency between each subsequent instruction. On Von Neumann hardware, this code would be executed efficiently, because instructions are fetched in program order. The

processor automatically fetches instructions in a sequence.

Figure 4.4.B is a pipeline diagram. It shows how the program would be executed on our pipeline, and reveals a problem with the initial dataflow implementation. Each column is a different cycle, so at each point in time we can see what stage an instruction is in. We can also see that each instruction takes 6 cycles to execute. In the first cycle, the pipeline begins fetching the first instruction. Even though the fetch width is 12, as mentioned above, only addi0 is fetched because the data has not been generated for the second or third instruction. This version of the pipeline does not know what the next instruction is going to be until the tokens are generated. It takes 6 cycles to execute each instruction, and each instruction waits until its parent instruction has been executed before entering the pipeline. Therefore, it takes 18 cycles to execute this program.

To alleviate this problem, we integrated a next-instruction prefetcher to the pipeline. When a token is processed, and the instruction is fetched, the prefetcher automatically fetches the next instructions in memory. If a fetch request is made for address A, then the prefetcher also fetches the instructions at address A+4, A+8,A+12, etc. Essentially, the prefetcher guesses which instructions are going to be needed next in an attempt to hide the fetch latency between dependent instructions.

Prefetched instructions are speculatively fetched, decoded, and placed into a prefetch buffer in the match unit. When a token is generated, it is sent to the match unit to see if the instruction has already been fetched. If it has, then we can send the instruction to the next stage without repeating the fetch and decode stages.

Figure 4.4.C shows the pipeline diagram with prefetching enabled. It takes about half the number of cycles (10) to execute the program because addi1 and addi2 are fetched and decoded at the same time as addi0, hiding the latency. Instructions that are prefetched incorrectly are simply discarded.

Match Unit The matching unit is used to detect which instructions are available for execution. This module is significantly different from any other in Von Neumann architecture. Out-of-order cores utilize hardware to analyze dependencies between instructions and detect

instructions that are ready. Instead of a register file, the matching unit has a small cache of memory for storing data tokens, called matching space (also called token store). When both of an instruction's operand tokens have been generated and sent to the matching space, the instruction is "fired" and sent to the next stage to be executed.

The match space is implemented as a fully associative cache using the token's tag as the key. The size of the cache is set to 100K so we can test large benchmarks. A detailed analysis needs to be done to find the optimal size. There are two scenarios that result in an instruction being executed. If an instruction only has a single operand, then it bypasses the match unit and is sent directly to the execute stage. If the instruction requires two operands, then the first token is stored in the cache when it arrives. At some time in the future, the second operand token is generated and a match is detected because both tokens have the same tag (and are mapped to the same spot in the cache). The first token is removed from the cache and both tokens are sent to the next stage for execution. This is why the number of operands is limited to two, since more operands would complicate the matching process.

Figure 4.5 shows the internals of the stage. It is composed of a cache for storing and matching tokens, and a buffer for temporarily storing prefetched instructions. The only thing stored in the match unit is the data and the destination address. The tag is used to index the cache.

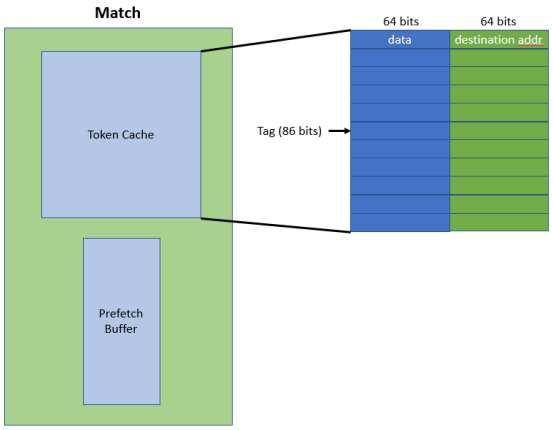

Figure 4.5: Match stage with token cache and fetch buffer

Execute The execute stage is largely similar to ones found in Von Neumann architecture. There are a number of functional units for issuing instructions in parallel. Each functional unit is pipelined, allowing them to work on multiple instructions.

An instruction arrives to the execute stage with both of its operands (because a match has already been determined). The instruction is immediately issued to one of the available functional units. The only deviation from Von Neumann architecture is during the write back process. The executed instruction produces between 1 and 3 result tokens. Instead of writing back to a register file, these tokens are sent back to the match stage. The circular pipeline is then repeated with the result tokens.

The issue width of the execute stage is four, so four instructions can be issued to functional units simultaneously. Similarly, four tokens can be produced every cycle.

4.3 Hybrid Microarchitecture

The hybrid CPU is shown in Figure 4.6. The out-of-order core and the dataflow core communicate either through direct communication (message passing) or through a shared memory system. The shared memory is used when large amounts of data needs to be communicated. For example, if an array is needed, the OoO model will initialize the array and then pass the address of the array to the dataflow model with direct communication. Currently, only one core is active at one time. While one core is executing, the other is idle, waiting for the active core to hand over control. It could be possible in the future to explore parallel execution.

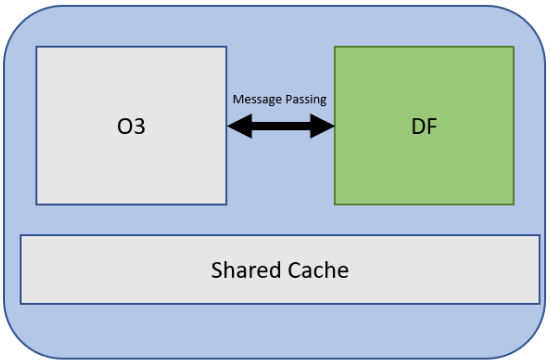

Figure 4.6: Hybrid Dataflow/Von Neumann

5 Methodology

This section describes the experimental and evaluation methodology to evaluate the proposed dataflow sub-ISA. We built a set of microbenchmarks to evaluate the sub-ISA and its microarchitecure implementation. Microbenchmarks are used to create common coding structures in order to evaluate the different cases where the dataflow ISA is superior to Von Neumann. This chapter also discusses the simulation models and their features, the process of compiling benchmarks using the dataflow sub-ISA, and finally the benchmark evaluation process.

5.1 The Gem5 Simulation

The gem5 simulator [21] is a computer architecture and microarchitecture simulator that is extensively used in research. It is a popular simulator because of its flexibility and because it supports many different ISAs (x86, arm, alpha, mips, RISC-V) and CPU models (atomic, in-order, out-of-order) [22] as shown in Figure 5.1. The Atomic simple model is a single-cycle CPU that is used for functional simulations. This model is used to warm-up the cache and branch predictors and to fast-forward to specific regions of interest. The gem5 simulator also contains an out-of-order CPU model called O3. The O3 model is often used to simulate the state-of-the-art processors in the gem5 simulator. We added a dataflow model to the gem5 simulator to test our new sub-ISA. Additionally, we developed a framework for explicit dataflow execution in gem5, which previously could only simulate Von Neumann architecture.

The gem5 simulator is flexible because of its polymorphic design. Different layers of abstraction are used to separate the ISA from the CPU model. Each CPU is an abstract model that makes no assumption about the underlying ISA. This allows different ISAs to be swapped

without changing the CPU model.

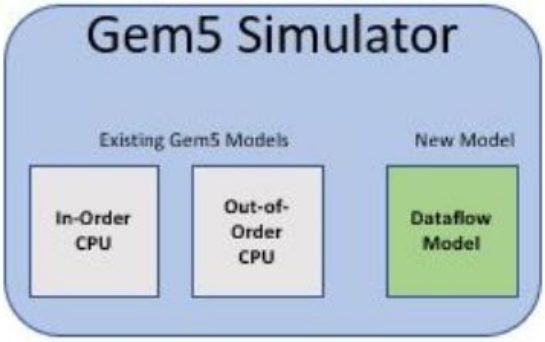

Figure 5.1: The gem5 simulator

However, gem5 still assumes Von Neumann architecture. Pure dataflow execution is not just a matter of created an additional CPU model, because even though the CPU models and ISAs are abstractions, they still make certain assumptions based on features that are common to all Von Neumann architectures. The two biggest challenges to dataflow execution:

- Built into each CPU model is the assumption of fetching driven by a PC.
- Built into the ISA structure is the concept of source and destination registers.

Rather than starting from scratch, our pipeline is a construct of different pipeline stages and constructions taken from existing gem5 models. This sometimes causes problems because it is based on Von Neumann assumptions. We analyze the pipeline to verify that the it is executing in a typical dataflow fashion. One example are the functional units. They were stalling the pipline unnecessarily because they were built in gem5 to detect dependencies between instructions and stall when there is a conflict. This is necessary for Von Neumann execution, but not for dataflow. We removed this problem from the simulator. A similar problem was detected with the Load/Store queue, but this problem is not yet solved due to the size and complexity of gem5.

The PC problem was fixed by adding a token queue to the fetch stage. The PC still exists, but rather than incrementing the PC, the address is taken directly from incoming tokens. Each cycle, a token is taken from the queue and the destination address is placed into the PC before a fetch request is made. The cache returns the instruction at that address. This creates dataflow fetching with very few modifications to the simulator.

The register file problem was overcome with a similar method. The gem5 simulator defines an ISA through switch statements. This statement defines the instruction behavior for each opcode, including source and operand registers. Our challenge is how to use this format to simulate a design without a register file? We solved this problem by creating virtual registers for dataflow instructions. We added 6 registers: operand0, operand1, destination0, destination1, destination2, and result. These registers are used to temporary store data before being transformed into data tokens. Again, a typical dataflow instruction executes by taking data from two operand tokens and producing a result token. When an instruction is executed, the required operand data is loaded in the two operand registers. The instruction executes and places the result and destinations into the corresponding registers. The simulator then uses these registers to create up to three data tokens.

5.2 Pipeline Viewer

Another useful tools is the pipeline viewer. Just looking at the output of a program is not very useful for debugging or analysis. A pipeline diagram tells us when an instruction enters the pipeline and the number of cycles that it stays in each stage. the gem5 simulator contains a tool that builds a pipeline diagram, but it only supports the o3 model. As such we implemented a tool that allows us to view the dataflow pipeline diagram. The pipeline diagram is constructed using debug print statements output from the simulator. Whenever an instruction enters a stage in the pipeline, this event is logged. The pipeline viewer reads the log file and constructs a visual representation of the pipeline. Figure 5.2 shows an example of the pipeline viewer output. It displays the program counter, operation, and pipeline diagram for each instruction.

```
10324_0  df_cpy          |F|F|F|F|D|D|M|-|
10328_0  df_cpy          |F|F|F|F|F|D|D|M|-|
1032c_0  df_cpy          |F|F|F|F|F|D|D|M|-|
```

Figure 5.2: Example PipeView output

5.3 Compilation Process

One challenge of computer architecture research is creating a compiler. Compiling for dataflow machines is especially challenging because traditional software assumes a Von Neumann CPU model. One option is to use special programming languages that are more suitable for dataflow architectures. The Manchester [11] machine was programmed using the ID programming language. This is a single-assignment language, meaning that each variable is generated once and cannot be modified. This restriction makes it easier to see true data dependencies between instructions, making it easier to compile for a dataflow architecture. We wish to balance the ease of compiling ID code while still being able to compile normal c code. Thankfully, tools exits which makes this practical. We use LLVM [23], a state-of-the-art compilation toolchain.

LLVM compiles C/C++ code down to an intermediate representation (IR). This language is ISA independent. While it assumes the use of some registers, each register is only initialized once, and never modified. LLVM IR is a single-assignment assembly language [23]. Essentially, LLVM provides a dataflow analysis of each instruction in a program, and our compiler uses this dataflow analysis to construct a dataflow graph.

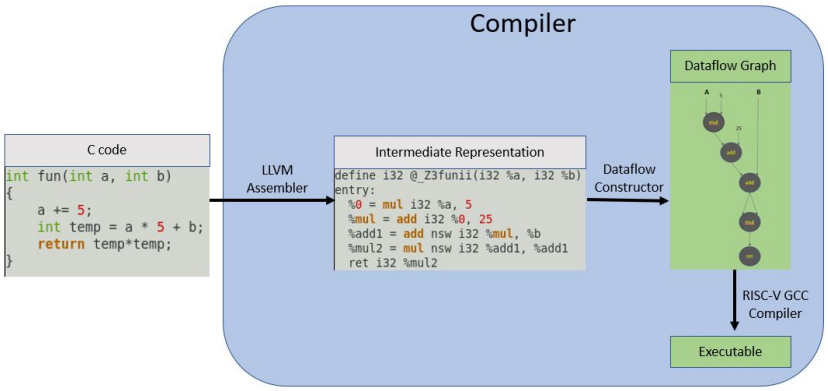

Figure 5.3: The dataflow compilation process

We created an LLVM module that creates the dataflow graph from the IR code. Each instruction in the IR code is converted into a dataflow instruction. The data dependencies between instructions are detected (by LLVM) and converted into explicit pointers. Once the graph has been formed, the next step is to generate the executable. Because there is no official RISC-V backend for LLVM, we can not use it for creating the executable. Instead, we added the

dataflow instructions to the RISC-V GCC assembler. We use inline assembly to add the output of our LLVM module directly to the source code. The RISC-V GCC then compiles the program, which is a mix of Von Neumann and dataflow instructions, into a single executable.

However, our compiler is limited because it does not correctly find all of the data dependencies. It relies on LLVM to extract all of the data dependencies and translate them into a dataflow graph. The compiler works for simple programs, but it fails to extract all of the dependencies accross nested loop iterations. LLVM does provide loop dependence analysis, but we have not correctly integrated this into the graph generator. To compensate for this, we correct the missing dependencies by hand.

5.4 Benchmark Selection

Because the compiler has yet to be completed, and dataflow graphs must be manually completed by hand, this work focuses on writing microbenchmarks that investigate specific properties of the dataflow architecture, determining its improvements and bottlenecks. This section describes the different programs, the reasons for choosing them, and the overall testing methodology. The next chapter reveals the results of these benchmarks and discusses the improvements and bottlenecks.

The difficulties of computer architecture research include the design of not only a simulation model, discussed in the previous section, but also the design or extension of a compiler that supports the new architecture we have designed. The problem is compounded because we are compiling standard C/C++ code to a non-traditional ISA.

Without a complete compiler, we focused on exposing the benefits and problems with the ISA through the implementation of microbenchmarks. Each microbenchmark exposes a different characteristic or flaw with our design and the dataflow model in general. Three microbenchmarks were implemented: sum array, indirect sum, and matrix multiplication. Appendix C shows the source code for the different microbenchmarks.

The sum array represents a simple for-loop implementation. Loops are ubiquitous in programming, so it is important to see how the dataflow version implements looping. This

program also represents a regular application–an application that accesses memory in a predictable manner.

Indirect array is similar in structure to the sum array, but does not access the array sequentially. Instead it accesses random elements of the array. Applications with irregular memory access patterns, also known as irregular applications, make it difficult or impossible to predict what location in memory a program is going to access next. We are interested to see if dataflow can outperform Von Neumann for this type of application.

Matrix Multiplication was chosen because it has more complex looping structure. Multi-nested loops are common in programming, so it is important to know ow well dataflow performs when executing complex control structures. This can give us insights on how to imporove the dataflow implementation in the future.

5.5 Benchmark Evaluation

The microbenchmarks were analyzed using the following procedure:

1. Compile for both dataflow and Von Neumann
2. Set checkpoint at begging of region of interest
3. Run both versions of the program separately in Gem5
4. Verify function equivalence
5. Use gem5 statistical output to measure speedup and reasons for speedup

The gem5 simulator is complex and it would take a very long time to simulate an entire application, which makes it impractical to simulate an entire benchmarks with large input arrays. Instead, a region of interest is determined and a checkpoint is created at the beginning of the region. Each region is compiled twice; first using instructions from the standard set, and again using the dataflow sub-ISA. Both versions of the code are run separately for comparison.

First, we compare the functional output of the simulator to see if both versions are functionally equivalent. This is done by running the program with various different input combinations.

If the dataflow version produces the same results as the Von Neumann, we conclude that they are functionally equivalent.

Next, we evaluate the performance of the two experiments. The gem5 simulator outputs numerous statistics. The following stats are used in this work:

- sim inst: total number of instructions simulated
- cycles: total number of cycles simulated
- dcache miss rate: percentage of memory instructions that resulted in a L1 cache miss
- L2 cache miss rate: percentage of memory instructions that resulted in a L2 cache miss

Because both the dataflow and O3 models use the same clock frequency, the number of cycles is used to calculate the execution time and thus the speedup or slowdown over the O3 model. The other statistics are used to determine why there is a difference in performance between the two models. The number of simulated instructions is used to detect if there are any overheads associated with the dataflow sub-ISA. Finally, the cache miss rate for L1 and L2 are used to determine if the memory access patterns are different between the O3 and dataflow architectures.

Next we draw conclusions about if we should use the dataflow sub-ISA for the given case. The question arises when to use dataflow instructions, and when to use instructions from the base ISA. One scenario is when dataflow it is difficult to compile due to the limitations of the compiler. If the program makes a call to a library compiled in the Von Neumann ISA, then we would not be able to use dataflow at this time. One option in the future is to recompile the entire library into the dataflow ISA. There are two choices after detecting bottlenecks in the sub-ISA: optimize the dataflow sub-ISA or hardware, or use the Von Neumann instructions.

6 Results

This chapter is split into three sections–each describing the results one of the microbenchmarks used in this work. The microbenchmarks are analyzed using the statistical output from gem5. Each microbenchmark is run on the dataflow model and gem5's out-of-order model, O3. The statistics used for analysis are normalized to the O3 value (so the O3 value is always 1). The source code for each microbechmark is attached to appendix C.

6.1 Case 1: Simple Loops and Regular Memory Access Patterns

Our first microbenchmark is a simple program that finds the sum of an array of 10 million integers. A single for-loop iterates over each element in the array, making the code relatively simple and, because the array is accessed sequentially, this is an example of a regular application. Because OoO processors are efficient at executing regular programs, a significant speedup is not expected. We calculate a speedup of approximately 1 (1.01x), assuming approximately the same number of instructions and similar cache statistics.

Figure 6.1 shows four statistics output from the gem5 simulator. While there are many more statistics available, these four are the most relevant for this analysis: number of instructions simulated, number of cycles the program took to execute, and the data cache (dcache) miss rate for L1 and L2. The left bar is the O3 result, and the right shows the same statistic when run on the dataflow model. For easy comparison, each value is normalized to the O3 result.

A 9% speedup over the OoO processor is detected. The cache miss rate helps us understand why–there are about 4x more cache misses on the OoO processor. We attribute this to the

fact that dataflow and OoO have different memory access patterns. In this case, the dataflow accesses the cache in a way that has better locality than the OoO. Therefore, the dataflow performance exceeded expectations.

Figure 6.1: Simulated instructions, number of cycles, and cache miss rate for the sum array program. Each value is normalized to the O3 value.

Here a situation where dataflow can provide similar performance to OoO: simple control structure and predictable data accesses. While it can even improve performance under some circumstances, we do not expect this to be true for all regular applications.

6.2 Case 2: Irregular Application

The second case is similar to the previous except it finds the sum of an indirect array, as shown in the C-code below. Rather than accessing the array sequentially, the index is loaded from a second array, which was created randomly. Essentially, this is the same program above but access the array in a random order. Irregular applications have memory access patterns that make it difficult or impossible to predict where the next memory access is going to be, resulting in a high number of data cache misses.

Dataflow has the potential to improve the performance of irregular applications. Out-of-order are able to handle regular applications but have trouble with irregular programs because of the limited window of instructions. Irregular applications cause the pipeline to stall if a large number of load instructions fill the ROB or the LSQ. The dataflow model, on the other hand, is less likely to stall the pipeline because it has a bigger window of instructions. Therefore, the dataflow sub-ISA should be able to tolerate irregular applications better than the O3 model.

However, the performance did not match expectations. Figure 6.2 shows the same metrics used in the sum array analysis. Looking at the number of cycles, the dataflow version takes about 2.4x longer to execute than the O3 model. What is unusual is that the slowdown is not consistent with the other statistics. When the program was initially run, the number of dynamic instructions was much higher for the dataflow program. We attributed this to be the reason for the slowdown. The next step was to perform loop unrolling to increase the basic block size. After optimization, the number of simulated instructions was lowered significantly, and now it is approximately the same as the OoO model, but the performance did not improve. Additionally, the two programs have almost identical dcache and L2 cache miss rates. The other statistics from the simulator show the same–little difference between the OoO and dataflow, yet there is a significant slowdown.

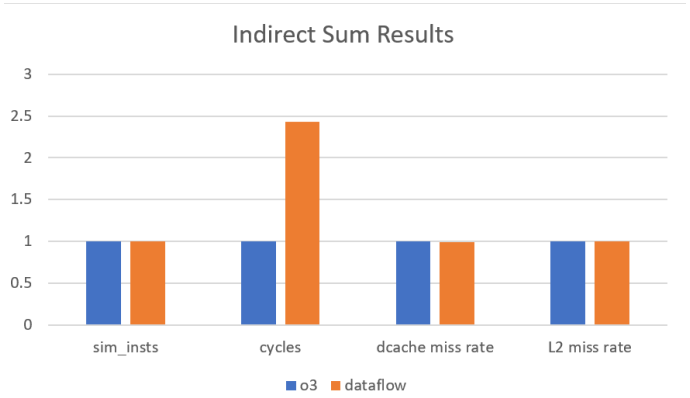

Figure 6.2: Simulated instructions, number of cycles, and cache miss rate for the indirect array program.

The problem was detected using our pipeline viewer and the GDB debugger to view the behavior of the pipeline. The dataflow pipeline is stalling unnecessarily. We attribute this to a flaw in the simulator caused by the contention between Von Neumann and dataflow execution. The load-store queue (LSQ) temporally buffers all memory instructions before they are executed. The size is set to 72 entries, but GDB reveals that the LSQ stalling the pipeline after only two load instructions. The problem derives from the fact that memory disambiguation is handled differently between the dataflow and Von Neumann ISAs. The LSQ must be modified so it can be adapted to dataflow execution. This a pitfall when trying to adapt a Von Neumann simulator to dataflow.

This problem was not observed in the previous case (sum array) because, being an irregular application, there are significantly more cache misses for the indirect sum, making the problem move noticeable. By our estimation, from analyzing the cache miss rate, program structure, and actual size of the LSQ, there should be a 1.2x speedup.

6.3 Case 3: Multi-Nested Loop

Our final case, matrix multiplication, is a regular application but uses three nested loops. Based on previous dataflow work, it is expected that this benchmark will have worse performance due to the control flow overhead of converting complex looping structures to a dataflow graph.

Figure 6.3 shows a 1.3x slowdown for this program. While the body of the loop is similar in structure to Von Neumann, there is a significant increase in the number of control instructions (even after the hand optimization phase). There are about 1.29x more instructions for the dataflow version which correlates to the 1.3x slowdown. The overhead from excess control instructions is the cause the slowdown. The figure also shows significantly fewer dataflow cache misses, but because the L1 dcache miss rate is only 0.2% for the O3 model, this does not provide significant speedup. This case indicates that complex control structures should be avoided for dataflow execution.

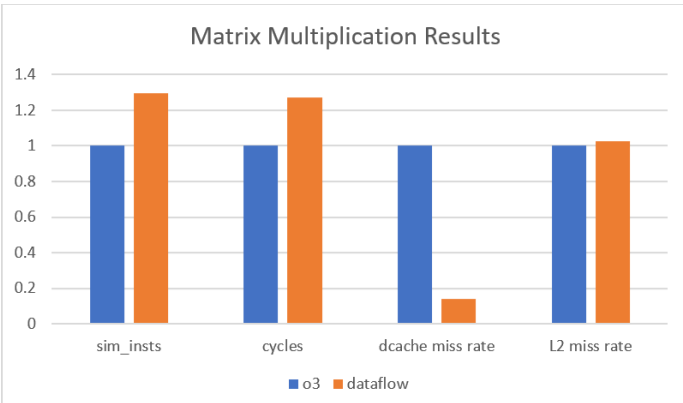

Figure 6.3: Simulated instructions, number of cycles, and cache miss rate for the matrix multiplication program.

There are two possible conclusions to draw: delegate this type of control structure to Von Neumann instructions, or find ways to optimize the code/hardware to reduce the control flow bottleneck.

7 Conclusion

This work implemented and evaluated a dataflow sub-ISA extension to the RISC-V ISA. Some cases were detected where dataflow execution provides better performance than the out-of-order (OoO) execution model, and other cases show a slowdown. As such, a hybrid dataflow/Von Neumann model that combines the benefits of both architectures is preferred. By extending a traditional Von Neumann ISA with specialized dataflow instructions, our microarchitecture model can improve performance of applications and even reduce their energy consumptino because the dataflow model is more energy efficient compared to OoO.

From our analysis, the dataflow sub-ISA should be used when the number of control instructions is low, or can be optimized away, the dataflow execution produces memory access patterns that result in lower cache miss rates, or the program is highly irregular. Conversely, the OoO architecture is better for regular applications that have complex control structures, such as multi-nested loops, and regular memory access patterns like the matrix multiplication.

A limitation of this work is the compiler, which limits our methodology and the scope of the research:

1. The benchmarks are limited to microbenchmarks. It takes too long to manually correct all of the mistakes for a program with hundreds or thousands of lines of code.

2. Without the optimization techniques found in most compilers, the dataflow code is less optimal than a Von Neumann version. We are forced to optimize the dataflow graphs by hand. This is difficult and time consuming for large benchmarks.

3. An improved compiler would allow for expiements that test different partitioning between normal RISC-V and the dataflow sub-ISA. This would make it easier to find the

regions where dataflow would normally be better. This research compiles a single region per program, and measured the speedup of that region alone.

7.1 Future Work

Below is a list of things that can be improved or added to the dataflow model that we plan to add in the future.

- We will explore options to remove the control overhead discussed above.
- The capabilities of the gem5 simulator need to be extended to make it easier to explore different models of computation like dataflow. This will fix problems with the simulator, including the LSQ issue discussed in the results section.
- RTL implementation for more accurate results.
- Experiment with different dataflow microarchitecture features.
- Measuring energy efficiency is outside the scope of this thesis. By removing complex hardware used by OoO processors–ROB, register renaming, etc.– dataflow processors have the potential to lower energy cost, while still providing improved ILP. Future work will compare energy consumption of the sub-ISA with the OoO model.
- Finally, we will extend the capabilities of the compiler, which will enable the execution of larger programs and experiments with different partitioning schemes between the dataflow and Von Neumann instructions.
- Rather than statically choosing the dataflow regions, A hybrid architecture capable of dynamically switching between the dataflow and Von Neumann ISAs could greatly improve performance.

Appendix A The Gem5 Simulator

The following are snapshots that showcase some of our modifications to the gem5 simulator.

```
if (inst->staticInst->getName() == "df_cpy") // if this is a copy
instruction, execute it here
{
    DPRINTF(Match, "found a cpy instruction, executing it here\n");
    DPRINTF(Match, "sending new instruction 0x%x: %s, binary=0x%x %lx\
n", pc, inst->staticInst->disassemble(pc), inst->staticInst->machInst,
t.context);
    execute_copy(inst, t);
    cpu.inst_count.total++;
    cpu.inst_count.copy++;
}
else if (is_monadic)   // if instruction only has one operand, send to
 execute stage
{
    op0 = t.data;
    op1 = 0;
    DPRINTF(Match, "0x%lx is monadic, skipping match unit: %lu\n", t.
destination, t.data);
    fired = true;
}
else if(lookup(context_tag) == false)  // if other operand is not in
match space
{
    DPRINTF(Match, "no match\n");
```

```
        // store token in match space
        match_space[context_tag] = data_in;
        if ( max_match_size_ < match_space.size() )
        {
            max_match_size_ = match_space.size();
            max_match_size = max_match_size_;
        }
    }
    else
    {
        DPRINTF( Match, "match: %d, %d\n", t.data, match_space[context_tag
    ].token.data );
        t1 = match_space[context_tag].token;

        fired = true;
        match_space.erase( context_tag );  // remove from match

        if (t.left_right == 'L')
        {
            op0 = t.data;
            op1 = t1.data;
        }
        else
        {
            op0 = t1.data;
            op1 = t.data;
        }
    }
```

Listing 1: Testing to see if an instruction is ready to be executed ("fired"). Copy instructions are executed immediately instead of being sent to the execute stage.

```
if(!token_queue.empty())
{
    DataToken t = token_queue.front();
    token_queue.pop();
```

```
    Addr pc = t.destination;
    if (pc)
    {
        DPRINTF(Fetch, "sending new fetch request for pc 0x%x with context
    %x\n", pc, t.context);
        fetchLine(pc, t, false);
        in_flight_fetch_requests++;
        prefetch_pc = pc; // start prefetching instructions at this addr
    }
}
```

Listing 2: The dataflow Fetch stage grabs a token from the queue, and fetches the corresponding instruction.

```
void
BaseHybrid::Toggle()
{
    switch (df_switch)
    {
        case runningO3:
            df_switch = runningDataflow;
            df_cpu->start();
            DPRINTF(Profile, "was in o3 for %lu cycles\n", (curCycle() -
    cycle_start));
            assert(df_cpu->pipeline->match.match_space.empty());
            DPRINTF(Hybrid, "size of match space: %d\n", df_cpu->pipeline
    ->match.match_space.empty());
            break;
        case runningDataflow:
            df_switch = runningO3;
            DPRINTF(Profile, "was in dataflow for %lu cycles\n", (curCycle
    () - cycle_start));
    };
    cycle_start = curCycle();
```

```
}
```

Listing 3: A simple switching mechanism for switching between OoO and dataflow execution. The tick function is called every clock cycle.

```
void
Pipeline::tick()
{
    cpu.stats.cycle_count++;

    execute.evaluate();
    match.evaluate();
    decode.evaluate();
    fetch.evaluate();
```

Listing 4: Snapshot from pipeline.cc which calls each of the pipeline stages every cycle.

```
0x2: decode OPCODE{
    format ROp {
    0x00: df_add({{
        df_dir = DIR0 | (DIR1 << 1) | (DIR2 << 2);

        df_dest0 = (DESTINATION0 > 0x3f) ? (DESTINATION0 | 0
    xfffffffffffffc0) : DESTINATION0;
        df_dest0 = (df_dest0 == 0) ? 0 : 4*df_dest0 + PC;

        if (!IS_IMMEDIATE)
        {
            df_dest1 = (DESTINATION1 > 0x3f) ? (DESTINATION1 | 0
    xfffffffffffffc0) : DESTINATION1;
            df_dest1 = (df_dest1 == 0) ? 0 : 4*df_dest1 + PC;

            df_dest2 = (DESTINATION2 > 0x3f) ? (DESTINATION2 | 0
    xfffffffffffffc0) : DESTINATION2;
            df_dest2 = (df_dest2 == 0) ? 0 : 4*df_dest2 + PC;
        }
        else
```

```
        {
            df_op1 = (DF_IMMEDIATE > 0x7fff)? (DF_IMMEDIATE | ~((uint32_t)
    0xffff)) : (uint32_t)DF_IMMEDIATE;
            df_dest1 = 0;
            df_dest2 = 0;
        }

        df_result = df_op0 + df_op1;
        }});

    0x01: df_call({{
```

Listing 5: Gem5 ISA description for dataflow add instruction. It calculates the result and destination address of the instruction. Each destination is cacluated by a sign-extended offset from the address of the current instruction (PC).

Appendix B LLVM

The following code is a section from our LLVM module that creates the dataflow graph. This function adds a data edge between two instructions. What complicates the program is if the case where the two instructions belong to different basic blocks. Then branch instructions must be inserted in between.

```
void add_edges(llvm::BasicBlock *bb, string src_id, llvm::Instruction *
    destination, string direction)
{
    Block *b = find_block(getId(&*bb));
    if (destination->getParent() == bb)
        b->edges += src_id + "->" + getId(destination) + direction + "\n";
    else if (llvm::BranchInst *br = llvm::dyn_cast<llvm::BranchInst>(bb->
    getTerminator()))
    {
        // insert branch instruction
        string br_id = getId(bb) + "_" + to_string(output_count++);
        b->instructions += br_id + ": br\n";

        llvm::BasicBlock *false_path = br->getSuccessor(0);
        llvm::BasicBlock *true_path = NULL;
        if (br->getNumSuccessors() > 1) true_path = br->getSuccessor(1);

        // false path
        if (is_dependent(false_path, destination->getParent()))
        {
            // add edge from src to branch
            b->edges += src_id + "->" + br_id + "R\n";
```

```
            // add edge from compare to branch
            b->edges += getId(br->getCondition()) + "->" + br_id + "L\n";

            add_edges(false_path, br_id+"[F]", destination, direction);
        }
        // true path
        if (is_dependent(true_path, destination->getParent()))
        {
            // add edge from src to branch
            b->edges += src_id + "->" + br_id + "R\n";

            // add edge from compare to branch
            b->edges += getId(br->getCondition()) + "->" + br_id + "L\n";

            add_edges(true_path, br_id+"[T]", destination, direction);
        }
    }
}
```

Listing 1: This function adds data edges between dependent instructions in the dataflow graph.

Appendix C Microbenchmark Source Code

The following programs are the microbenchmarks used in this work.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

unsigned size = -1;

int sum(int *array)
{
    int sum = 0;
    for (int i=0; i<size; i++)
        sum += array[i];
    return sum;
}

int main( int argc, char *argv[])
{
    srand(20);

    size = atoi(argv[1]);

    int *array = malloc( (unsigned long)size*sizeof(int) );
    assert( array );
```

```
    for (int i=0; i<size; i++)
    {
        array[i] = rand() % 2000;
        printf("%d: %d\n", i, array[i]);
    }
    printf("sum of array = %d\n", sum(array));
}
```

Listing 1: Source code for sum array.

```
#include <stdio.h>
#include <stdlib.h>
#include <random>

unsigned size = -1;

int *array;  // array of integers
int *idx;  // array of indices

long indirect_sum()
{
    long sum = 0;
    for (int i = 0; i<size; i++)
    {
        sum += array[idx[i]];
    }
    return sum;
}

int main( int argc, char *argv[])
{
    srand(20);

    size = atoi(argv[1]);

    std::default_random_engine generator;
```

```
    std::uniform_int_distribution<int> distribution(0,10*size);

    // create index table
    array = (int*)malloc( 10*size*sizeof(int) );
    idx = (int*)malloc( size*sizeof(int) );
    for (int i = 0; i<10*size; i++)
        array[i] = rand() % 2000;

    for (int i = 0; i<size; i++)
    {
        idx[i] = distribution(generator);
        printf( "array[%x] = %x\n", idx[i], array[idx[i]] );
    }

    printf( "sum = %ld\n", indirect_sum() );
    return 0;
}
```

Listing 2: Source code for indirect sum (C++).

```
#include "stdio.h"
#include "stdlib.h"

#define A_rows 100
#define A_cols 100
#define B_rows 100
#define B_cols 100

int A[A_rows][A_cols];
int B[B_rows][B_cols];
int result[A_rows][B_cols];

void matrix_mult(int A[A_rows][A_cols], int B[B_rows][B_cols], int result[
    A_rows][B_cols])
{
    for (int k=0; k<A_rows; k++)
```

```
    {
        for (int j=0; j<B_cols; j++)
        {
            int sum = 0;
            for (int i=0; i<A_cols; i++)
                sum += A[k][i]*B[i][j];
            result[k][j] = sum;
        }
    }
}

int main()
{
    srand(10);

    // initialize arrays
    printf("array A:\n");
    for (int i=0; i<A_rows; i++)
    {
        for (int j=0; j<A_cols; j++)
        {
            A[i][j] = rand() % 10;
            printf("%d ", A[i][j]);
        }
        printf("\n");
    }

    printf("array B:\n");
    for (int i=0; i<B_rows; i++)
    {
        for (int j=0; j<B_cols; j++)
        {
            B[i][j] = rand() % 10;
            printf("%d ", B[i][j]);
        }
```

```
        printf("\n");
    }
    matrix_mult(A, B, result);

    for (int i=0; i<B_cols; i++)
    {
      for (int j=0; j<B_cols; j++)
         printf("%d ", result[i][j]);
      printf("\n");
    }

   return 0;
}
```

Listing 3: Source code for matrix multiplication.

Bibliography

[1] K. A. Editors Andrew Waterman, ed., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017.

[2] J. B. Dennis, "First version of a data-flow procedure language," *In Proceedings of the Colloque sur la Programmation*, vol. 19, 362-376.

[3] J. E. Rumbaugh, "A data flow multiprocessor," *IEEE Computer*, vol. 26, no. 2, pp. 138–146, 1977.

[4] A. H. V. Data, "Dataflow machine architecture.," *ACM Computing Surveys (CSUR)*, vol. 18, no. 4, pp. 365–396, 1986.

[5] J. B. Dennis and G. R. Gao., "An efficient pipelined dataflow processor architecture.," *In Proceedings of Supercomputing*, pp. 368–373, 1988.

[6] R. Nikhil and G. Papadopoulos, "T: A multithreaded massively parallel architecture.," *In Proceedings of 19th International Symposium on Computer Architecture*, pp. 156–167, 1988.

[7] R. S. Nikhil and Arvind, "Can dataflow subsume von neumann computing?," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 262–272, 1989.

[8] G. Gao, "An efficient hybrid dataflow architecture model.," *Parallel and Distributed Computing,*, vol. 19, no. 4, pp. 293–307, 1993.

[9] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, vol. 43, pp. 298–310, June 2015.

[10] D. Burger, S. W. Keckler, K. S. McKinley, M. Moore, J. Burrill, R. G. McDonald, and W. Yoder, "Scaling to the end of silicon with edge architectures," *IEEE Computer*, vol. 37, pp. 44–55, July 2004.

[11] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, pp. 34–52, Jan. 1985.

[12] Arvind and R. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, pp. 300–318, Mar. 1990.

[13] G. Papadopoulos and D. Culler, "Monsoon: an explicit token-store architecture," *Proceedings. The 17th Annual International Symposium on Computer Architecture*, pp. 82–91, May 1990.

[14] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *ISCA '75 Proceedings of the 2nd annual symposium on Computer architecture*, vol. 3, pp. 126–132, Jan. 1975.

[15] V. Grafe, G. Davidson, J. Hoch, and V. Holmes, "The epsilon dataflow processor," *The 16th Annual International Symposium on Computer Architecture*, pp. 36–45, June 1989.

[16] V. Grafe and J. Hoch, "The epsilon-2 hybrid dataflow architecture," *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, pp. 88–93, Feb. 1990.

[17] R. Iannucci, "Toward a dataflow/von neumann hybrid architecture," *The 15th Annual International Symposium on Computer Architecture.*, pp. 131–140, May 1988.

[18] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin,

and S. J. Eggers, “The wavescalar architecture,” *ACM Transactions on Computer Systems (TOCS)*, vol. 25, May 2007.

[19] Arvind, R. S. Nikhil, and K. K. Pingali, “I-structures: data structures for parallel computing,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, pp. 598–632, Oct. 1989.

[20] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, “Tartan: evaluating spatial computation for whole program execution,” *ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, vol. 34, pp. 163–174, Oct. 2006.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. W. and, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, May 2011.

[22] A. Akram and L. Sawalha, “A survey on computer architecture simulation techniques,” *IEEE Access*, vol. 7, pp. 78120 – 78145, May 2019.

[23] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis and transformation,” *CGO ’04 Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, Mar. 2004.