WAVEFRONT SCHEDULING ON SUPERSCALAR PROCESSORS

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Joshua Scoggins

2012

SIGNATURE PAGE

THESIS:	WAVEFRONT SCHEDULING ON SUPERSCALAR PROCESSORS
AUTHOR:	Joshua Scoggins
DATE SUBMITTED:	Summer 2012
	Department of Computer Science

Dr. Daisy F. Sang
Thesis Committee Chair
Computer Science

Dr. Craig A. Rich
Computer Science

Dr. Gilbert S. Young
Computer Science

Abstract

One of the easiest ways to improve processor performances is by exploiting *instruction level parallelism* through the inclusion of multiple *execution units*. Each execution unit is capable of executing an instruction. By identifying and dispatching independent instructions it is possible to execute multiple instructions per cycle. The act of identification and dispatch, also known as *scheduling*, can be performed dynamically or statically.

Dynamically scheduled processors are known as *superscalar* processors. They use hardware to identify parallelism in a sequential program at runtime. The majority of processors from companies like Intel, AMD, and IBM are superscalar.

Statically scheduled processors shun the scheduling hardware found in dynamically scheduled processors to add more execution units. The task of scheduling is given to the compiler to perform during program compilation. This requires that the compiler implement *region scheduling algorithms* to identify parallelism in far larger spans of code than what is possible in hardware. Most statically scheduled processors are found in the realms of microcontrollers and video cards.

Wavefront Scheduling is a region scheduling technique developed by Intel for improving performance on their Itanium line of processors. Wavefront scheduling is different from other region scheduling algorithms in that it schedules instructions with respect to the other paths in a given region.

This thesis explores the viability of wavefront scheduling on superscalar architectures such as x86. Since superscalar processors schedule dynamically, what would happen if the compiler assisted the processor by prescheduling the program statically? The idea is that the processor would *rediscover* the parallelism already identified during compilation. This would theoretically allow the processor to execute more instructions per cycle because the work has already been done. It was also decided to implement wavefront scheduling in a novel fashion as an expert system in CLIPS which was integrated into the LLVM compiler infrastructure. The final result of this thesis is that applying wavefront scheduling to programs targeting superscalar architectures provides a speed boost as well as showing that an expert system can be used to optimize code.

List of Figures

List of Tables

Contents

Chapter 1

Background

1.1 Introduction

Most modern high-performance processors take advantage of a technique called Instruction Level Parallelism (ILP) to potentially reduce the running time of a given sequential program by simultaneously executing several instructions per clock cycle [1]. This potential is tempered by the following factors:

1. The amount of parallelism in a given program
2. The parallel capabilities of the processor
3. The ability to identify parallelism from a sequential program.
4. The ability to construct the best parallel schedule possible based on system constraints

The amount of parallelism in a given program is directly affected by the amount of dependent instructions within the program. A program with a high frequency of data dependent instructions will not gain nearly as much of a performance improvement as a program that performs scientific computations where many operations are independent of each other.

Even if the processor is capable of executing several instructions per cycle, the mix of instructions may still cause instructions that can be executed in parallel to be executed sequentially due to processor limitations. For example, let us say that a processor can execute an integer and a floating point operation per cycle. Sections of integer or floating point exclusive code will still cause sequential execution to occur. Thus for this theoretical processor to execute more than one instruction per cycle a program consisting of an even mix of floating-point and integer instructions are required to maximize performance of the given program. This is not only hard to do consistently but may not even be possible in some cases. Circumventing this issue requires that the compiler be able to identify parallelism from the given sequential program and then construct the best possible parallel schedule from this information.

The rest of this document will first review the hardware techniques available for identifying instruction level parallelism from a sequential program, followed by a brief introduction to the Intel Itanium 2 microarchitecture, the construction of a parallel schedule through the use of list scheduling and wavefront scheduling; and finally a proposal.

1.2 Identifying Parallelism in a Sequential Program

When it is possible, identification of parallelism within a sequential program consists of extracting parallelism and scheduling it for parallel execution. Performing this identification is done dynamically in hardware (Superscalar), statically by software (Very Long Instruction Word), or through a combination of hardware and software means (Explicitly Parallel Instruction Computing) [1].

1.2.1 Superscalar Architectures

Most modern computer processors are superscalar. This means that the hardware performs dependency analysis on the instruction stream at runtime to detect dependencies between a very small set of instructions. Instructions that are found to not have any dependencies are dispatched to an appropriate functional unit by the hardware scheduler. This has quite a large number of benefits. The biggest being that the programmer visible instruction set is separate from the internal instruction set. This is achieved through the use of a translation layer that translates the exposed instruction set into the internal instruction set of the processor. This abstracts away the need to know memory access times and most implementation specific features about a given architecture. This allows programs compiled for the externally visible architecture to run on all revisions of the architecture without the need for re-compilation.

The biggest problem with superscalar processors is that the dynamic nature of the hardware scheduler means that long spans of instructions can not be dynamically scheduled while maintaining sane transistor counts, thermal properties, or cycle times. This limits the identification of parallelism to a small set of instructions. The other problem with superscalar processors is increasing the number of functional units requires the scheduling hardware to be updated to take these new units into account. Too many functional units will prevent the hardware scheduler from taking complete advantage of all the available execution resources due to time constraint the scheduler is under [1]. Thus it is up the hardware designer to balance the number of functional units with the complexity of the associated hardware scheduler.

Even with the issues stated above the superscalar technique is the most common technique used to exploit ILP. There are several ways to implement hardware scheduling with the oldest, and most relevant being scoreboarding.

Scoreboarding

In a serial CPU each instruction is decoded and executed one after another. This is called in-order execution. A system that executes out-of-order must decouple decoding and dispatching. The instruction decoder is continually decoding instructions to be executed which is then passed off to the dispatch unit. The dispatch unit will start the instruction even in the face of required data dependencies. This allows the processor to continue to execute instructions even when multi-cycle instructions such as a multiply or divide are dispatched. The problem with out-of-order execution is that it requires hardware to keep track of data dependencies to determine what instructions can be safely dispatched out-of-order.

Scoreboarding ensures this safety by keeping track of all instructions that are currently in flight through the use of a logging system. Instructions are dispatched to the pipeline when it is found that the instruction has had all of its dependencies met and there is a functional

[1] Usually the hardware scheduler has maybe 5 clock cycles to perform analysis in!

unit available for executing the instruction. This technique was first implemented on the Cray CDC 6600 released in 1965. The scoreboard takes instructions that are decoded in order and runs them through four stages:

1. Issue
2. Read Operands
3. Execution
4. Write Result

In the *issue stage*, the processor checks, and records, what registers the instruction reads from and writes to. If a write after write (WAW) data hazard is found between this new instruction and an instruction already in execution then the newer instruction will be stalled until the older instruction has finished writing back to the associated register. Once this has occurred the newer instruction may still be stalled if there is not an appropriate functional unit available.

Once the instruction is deemed *issue ready* and has been bound to an appropriate, and free, functional unit it gets passed to the *read operands stage*. In this stage the processor stalls the instruction until the instructions operands are ready. This prevents read-after-write (RAW) data hazards from occurring because the processor will not pass the instruction to the next stage until the associated registers have actually been written to.

The third stage, *execution*, actually executes the target instruction using the functional unit the instruction was bound to in the *issue* stage. Reaching this stage also means that the source operands are ready.

Once the operation finishes execution it is passed to the final stage, *write result*, where the result of the operation is written back to the target destination register. The processor will stall the write if it finds that instructions need the value currently stored in the target register. This prevents the occurrence of write-after-read (WAR) hazards.

While scoreboarding is an effective method of exploiting instruction level parallelism, it can cause the processor to completely stall when it finds that there is not a functional unit free. This will bring the out-of-order execution to a halt even though there may be newer instructions that could be safely executed. To solve this problem a processor must use Tomasulo's algorithm instead. However, for the sake of brevity this method will not be discussed.

1.2.2 Very Long Instruction Word

Very Long Instruction Word (VLIW) exploits ILP by using a purely software approach to parallelism identification and schedule construction. The idea behind VLIW is that the area consumed by a hardware scheduler could be better used by including more functional units and registers. These increases allow the processor to execute more operations per cycle and keep more data locally within the processor which also increases the ability to execute more instructions per cycle. The term VLIW refers to the fact that processors of this type operate on groups of instructions bundled into very long instruction words. These instructions encode not only the operation but also which functional unit the instruction should be dispatched to. This change is necessary to ensure that the large number of functional units can be continually fed with instructions instead of waiting for enough single instructions to be built up to perform a dispatch. Since VLIW uses static scheduling it is up to the compiler to perform all of the features of a hardware scheduler including stalling. However, this approach does give the scheduler far more time to analyze far larger sections of a program for parallelism. This requires a compiler writer to be intimately familiar with how the architecture works. This includes information such as memory access times, functional unit count, register file layout, and even pipeline latencies[2].

This kind of knowledge is very implementation specific but has the benefit that improvements to the compiler will improve overall system performance. It also means that poorly implemented compilers will generate low quality programs. This software approach also has the unfortunate side effect that if the VLIW philosophy is not completely understood by the compiler writer then they will tend to only implement the minimum amount of scheduling required to extract some amount of performance out of the processor. This relegates the implementation of more advanced scheduling algorithms to a later version or not at all [2]. The other problem with the purely software approach is that programs are not binary compatible. This creates the need to recompile a program for each supported VLIW architecture and generation. The need to re-compile programs to reflect changes across the generations of a VLIW architecture makes it nearly un-maintainable for a commercial software vendor that supports multiple architectures. Either the vendor has to ship the source code of their program to be compiled by the user on their machine or the vendor must include every possible version with their installation media. This is a deal breaker because neither version will allow vendors to provide high quality service. These problems have relegated VLIW to the areas of embedded systems and high performance graphics cards where the need to recompile code to take maximum advantage of the hardware is not only commonplace but required for optimal performance.

1.2.3 Explicitly Parallel Instruction Computing

Explicitly Parallel Instruction Computing (EPIC) architectures exploit ILP through a hybrid software/hardware approach. It is a hybrid design that combines the massive parallel capability of VLIW with the compatibility and abstraction of superscalar. Research into EPIC was started in the late 1980s by HP trying to find a new way to take advantage of ILP without having to resort to increases in clock speed, pure dynamic scheduling, or improvements in branch predictor accuracy. Extracting performance improvements from these areas were seen to eventually succumb to diminishing returns. The massive parallelism of VLIW was seen as the way forward but the logistical problems of such a design was seen as a very large problem. The research put into EPIC came up with the following tenants:

1. The processor is a target for a compiler
2. The compiler is responsible for extracting ILP out of a sequential program
3. The compiler does not need to know the exact number of functional units as dispatch will be handled by the hardware.
4. The processor will operate on fixed long instruction words called bundles
5. Programs compiled for older versions of an EPIC processor will be binary compatible across generations.

6. Memory access latencies are abstracted away by the processor

7. Parallelism is described explicitly

8. The hardware should provide features to assist the compiler in maximizing ILP.

These tenants are quite broad but are a amalgam of features taken from both VLIW and superscalar philosophies. It could be inferred that EPIC is really just a VLIW processor with superscalar elements. The problem with this statement is that it is not very accurate. EPIC is very different from VLIW in many critical areas including: instruction encoding, how parallelism is described, a fully predicated instruction set, direct control of the branch predictor, and the abstraction of functional unit counts and memory access times. Understanding how these different features work is best explained through an understanding of the Intel Itanium 2 microarchitecture.

1.2.4 Itanium 2: An EPIC Implementation

The Intel Itanium 2 is an in-order pipelined EPIC architecture equipped with a unique register file, instruction layout, parallelism description, an interesting dispatch model, and a grab bag of hardware features including predication and speculation.

The Register File

The register file of the Itanium 2 microarchitecture consists of:

- 128 64-bit general purpose registers
- 128 82-bit floating point registers
- 128 64-bit application registers
- 64 One-bit predicate registers
- Eight 64-bit branch registers

This wide selection of registers ensures that a large amount of data can be located on the processor at any given point in time. This reduces the need to continually perform memory reads and writes which can potentially reduce the number of data dependent instructions and thus improve ILP. However, the layout of the register file is only one part of the uniqueness of the Itanium 2.

Bundles

Itanium 2 uses fixed size bundles as a way to describe groups of instructions to be executed in parallel by the processor. These bundles are 128-bits in length and consist of three 41-bit instructions and a five bit type code[3]. These bundles are what the Itanium 2 *front-end* operates on in order to provide binary compatibility across different generations of the Itanium architecture. However, the actual ILP is described through another concept called the instruction group.

Instruction Groups

In a VLIW architecture, instructions are encoded into very long instruction words which describes the ILP of the current cycle. Thus it is really important to maximize the number of instructions executed within each very long instruction word such that the word does not contain WAW, RAW, or WAR dependencies.

Instead of very long instruction words, Itanium 2 uses the concept of an *instruction group* to explicitly describe ILP. An instruction group is defined as being a group of instructions that do not contain any WAW or RAW dependencies[3]. These instruction groups do not have a fixed size and are delimited through the use of an explicit stop[2]. These explicit stops cause a one-cycle delay to occur which will cause operations currently being executed to either complete or progress to a point where the result can be read by dependent instructions[3]. It is important to note that instruction groups consist of bundles and the explicit stop is encoded into specially typed bundles that allow one bundle to complete and start the first instruction of the next bundle immediately after the explicit stop is fired.

An instruction group is described through the following example:

```
ld8 r34=0xfded
ld8 r35=0xafded
```

[2]The explicit stop is denoted in Itanium assembler as ;;

```
;;
add r36 = r34,r35
;;
```

In this example there are two instruction groups where the first instruction group loads two values from memory and the second instruction group adds the two values together. The idea of describing ILP through instruction groups is a very novel concept that requires a different method of execution and dispatch.

Execution Model

The Itanium 2 processor divided into two major sections: the *front-end* and the *back-end* [3]. The front-end is tasked with instruction dispatch and the back end is responsible for handling WAR data hazards and instruction execution.

The front-end is where bundles are queued until it is possible for execution[3]. The Itanium 2 microarchitecture is capable of operating on up to two bundles per cycle[3]. The bundles are decomposed by the front end and dispatched to an appropriate *execution unit*. This continues to occur until the processor is unable to dispatch instructions due to there not being any free execution units of the given instruction's type. When this happens, the processor stalls those unfulfilled instructions and dispatches them in the next cycle[3].

An execution unit is a layer of indirection added to abstract the nearly 30 functional units in the Itanium 2 microarchitecture from the instructions themselves[3]. This means that each execution unit maps to more than one functional unit on the backend[3]. There are four types of execution units in the Itanium 2 microarchitecture and seven instruction types in the Itanium 2 instruction set[3]. These types are described in Table 1.1.

Class	Maps To	Description
Arithmetic (A)	I or M	Will dispatch to I or M type execution units
Integer (I)	I	Standard integer based operations
Memory (M)	M	Standard memory based operations
Floating Point (F)	F	Standard floating point based operations
Branch (B)	B	Standard branching operations
Extended Branch (X)	B and M	Consumes two slots of a bundle. 64-bit branching

Table 1.1: Itanium instruction types and associated descriptions

When the execution unit has dispatched the given instruction it becomes part of the processor's scoreboard. As stated above, an instruction group consists of instructions that have no RAW or WAW dependencies. The third data dependency type, WAR, is handled internally by the processor through the use of this scoreboard[3]. A scoreboard was added because it is actually quite difficult for a compiler to know if a data element will be in the processor's level one data cache when it is needed. This lack of information requires that the compiler use very conservative estimates with respect to the act of loading data from main memory. The use of a scoreboard removes this issue by allowing the hardware to stall instructions if it is found that the data from the given memory address is not in the level 1 data cache. This stall will cause all dependent instructions to stall as well. These instructions will only be dispatched once all prior instructions contained in the scoreboard are dispatched. Ordering these instructions is implicitly maintained due to the in-order design of the Itanium 2 microarchitecture[3]. Instructions that are a part of the group of dispatched instructions are also marked as stalled to ensure that any potential unknown dependencies are met[3].

The final thing to note is that all functional units contained in the back-end are *fully dispatched*[3]. This means that once the functional unit finishes computing the current instruction it is immediately available to every other functional unit. This feature allows for registers to be updated before the contents are written back to the register file. Furthermore, the Itanium 2 microarchitecture has fixed cycle times for all of its instructions[3]. This is defined by the table contained in Table 1.2.

Instruction Type	Cycle Time
Floating Point	4
Extended Branch	2
Memory	1
Integer	1
Branch	1
Arithmetic	1

Table 1.2: List of cycle times for given instructions

The Itanium 2 microarchitecture provides several features to assist the compiler in performing static scheduling. These are in the form of predication and speculation.

Predication

Itanium 2 features 64 one-bit predicate registers that enable conditional execution to occur on a per-instruction basis[3]. These predicate registers are also used to reduce the number of branch instructions. Instead of having separate branch instructions for operations like *branch if greater than* or *branch if less than*, the predicate registers are populated with this result through the use of *cmp* instructions. Each of these cmp operations store their results into two predicate registers. The first register stores the actual result of the comparison while the second register stores the inverse. This provides an efficient representation of two sides of a branch without having to resort to branch statements. This means that both sides of a branch can be viewed as normal instructions and can be scheduled together. This is called *control speculation.* The only other architecture that uses predication in any significant fashion is the ARM architecture where it is used to compress simple branch statements.

Data Speculation

Data speculation is another unique feature provided to the compiler by the Itanium hardware. It is designed to allow the compiler to schedule instructions across conditional branch boundaries. This kind of data speculation is done in many different architectures but Itanium provides the feature in hardware. When executing a non-reentrant function it is necessary to check to make sure that the data that is being referenced from an external source is of the correct value. If the value has been modified by another function since the load then it is necessary to jump into code designed to ensure correctness. Normally, these kinds of checks are handled entirely in software. Itanium provides a mechanism in the form of *ld.a* and *chk.a*[3]. A register is loaded using the *ld.a* instruction. This has the effect of loading the address of the target value into the Advanced Load Address Table (ALAT). The program the continues to execute instructions as normal. If the register associated with the advanced load is used for something else then the entry in the ALAT will be removed. Once the register referred in the advanced load needs to be used it is necessary to call *chk.a*. If the advanced load entry is still located in the ALAT then the load is finished and the value is placed in the register referred to by the *ld.a* instruction [3]. If the entry is not in the ALAT, then control is passed to a provided address in the program to perform the reload operation[3]. Once it is done then control is passed back to the point where the *chk.a* occurred.

This feature allows a compiler to load registers way in advance. The idea behind this is that the compiler believes that one path is going to be taken over the other and as such it is proper to start the act of loading values into memory as early as possible. The only downside with data speculation is that it requires the use of profiling to figure out what instructions should be speculated on[3] [2]. Because of this, it is a good idea to try to convert as much data speculation into control speculation [4]. Now that all three types of parallelism identification have been discussed it is now possible to discuss schedule construction.

1.3 Constructing the Best Possible Schedule

In compiler optimization theory a program is broken into a series of basic blocks which are a sequence of instructions where control enters the top of the block and leaves through the bottom of the block [1]. These basic blocks make up what is known as a control flow graph which describes the potential *flow* of a given program as a graph. Representing a program in this way has many advantages. First, the division into basic blocks means that local basic block optimizations such as dead code elimination can be applied. Second, structures such as loops can be described easily as a function of the graph itself. Third, the *shape* of the graph can also be used to apply optimizations and glean extra information about the program that would not be easy to figure out if the program was not represented as a graph. One of the most important pieces of information that is gleaned from the shape of the program is the list of dominators of a basic block[1]. A block A dominates B if the only way to get to B is through A. Thus a basic block is partially dominated if has more than one predecessor. This information is very useful in the process of schedule construction.

Constructing the most optimal schedule for a basic block is actually NP-complete. This does not prevent basic block scheduling from being useful because most basic blocks are quite small and thus it is possible to use a simple scheduling technique known as list scheduling to great effect [1].

Constructing a Data-Dependence Graph

Before schedule construction can actually begin it is necessary to create a data-dependence graph for a given basic block. This graph G consists of a set of nodes N, where each node is a machine instruction, and a set of edges E where each edge represents a data dependence between different instructions [1]. Constructing G consists of the following steps:

1. For each operation n in N, create resource reservation table RT_n which contains all of the data read from and written to for operation n. Also encode what functional units will be reserved by n.

2. For each edge e in E, label e with a delay d_e signifying that the destination node must be issued no earlier than d_e clocks after the source node is issued. When dealing with VLIW instructions it is important to note that the second step is extremely critical to ensure proper system performance. On Itanium, and other EPIC based systems, the need to stall waiting for memory loads is handled by the hardware.

These steps will create a data-dependence graph that describes the order of execution and what data will be required for each instruction. When the construction is finished it is possible to apply list scheduling.

List Scheduling

List scheduling works by visiting the nodes of the basic block in *prioritized topological order* [1]. It computes the earliest time slot in which each node can be executed based on the constraints imposed by its data requirements and the requirements of the previous nodes. Then the resources needed by the node are checked against a resource-reservation table constructed earlier that collects all the resources committed so far. The node is scheduled in the earliest time slot that has sufficient resources [1]. The algorithm for list scheduling is defined as follows in [1].

INPUT: A machine-resource vector $R = [r_1, r_2, \ldots]$, where r_1 is the number of units available of the ith kind of resource, and a data-dependence graph $G = (N, E)$. Each operation n in N is labeled with its resource-reservation table RT_n; each edge $e = n_1 \to n_2$ in E is labeled with d_e indicating that n_2 must execute no earlier than d_e clocks after n_1.

OUTPUT: A schedule S that maps the operations in N into time slots in which the operations can be initiated satisfying all the data and resources constraints.

METHOD:

```
RT = an empty reservation table;
for each $n$ in $N$ in prioritized topological order do
  // Find the earliest time this instruction count begin, given when its predecessors
  started
  $s = \max_{e=p \to n \in E}(S(p) + d_e)$;
  while there exists $i$ such that $RT[s+i] + RT_n[i] > R$ do
    $s = s + 1$ // Delay the instruction further until the needed resources are available
  end while
  $S(n) = s$;
  for all $i$ do
    $RT[s+i] = RT[s+i] + RT_n[i]$
  end for
end for
```

Prioritized Topological Order

It is important to note that list scheduling does not backtrack and schedules each node exactly one time [1]. This requires that nodes be selected on a heuristic priority function based on which one is ”ready” next. There is not a single heuristic priority function that works for all cases. Instead the priority function must be constructed based on what is the priority for the schedule[2]. Here are some observations about possible prioritized orderings as described in [1] and [2]:

- Without resource constraints, the shortest schedule is given by the *critical path*, the longest path through the data-dependence graph. A metric useful as a priority function is the *height* of the node, which is the length of a longest path in the graph originating from the node.

- On the other hand, if all operations are independent, then the length of the schedule is constrained by the resources available. The critical resource is the one with the largest ratio of uses to the number of units of that resource available. Operations using more critical resources may be given higher priority

- Finally, one can use the source ordering to break ties between operations; the operation that shows up earlier in the source program should be scheduled first.
- When dealing with embedded systems the target will most likely be energy efficiency or code size.

In essence, list scheduling is dependent on *what* the compiler writer is wanting to accomplish. This makes list scheduling very flexible but also difficult to grasp if this fact is not described. However, list scheduling can fail to find any instruction level parallelism within a basic block. Thus region scheduling algorithms were introduced to extend the size of the scheduling area beyond a basic block.

Region Scheduling Algorithms

Region scheduling algorithms were developed with the advent of VLIW processors to increase the size of the scheduling area beyond a basic block to detect more instruction level parallelism than is possible using basic block scheduling alone[2]. These algorithms operate on *regions* of code which is a set of basic blocks that has one entrance and one exit. This definition is very similar to that of a basic block and in fact it can be said that a basic block is actually a degenerate case of a region *compilers*.

Region scheduling algorithms are designed to allow scheduling algorithms, such as list scheduling, to be applied across multiple basic blocks. This is possible because region scheduling algorithms turn spans of basic blocks into a really long basic block through the use of rules that govern the size, shape, and content of these regions. There are many different region scheduling algorithms and each takes a different approach to scheduling instructions across multiple basic blocks.

Most of these algorithms rely on the collection of branch statistics to select and modify a given region of code. The problem with this approach is that it is quite difficult to generate accurate branch statistics because the *intent* of the code can not be gleaned from the intermediate representation alone. For instance, during profiling it seems that taking a branch in one direction will cause a divide by zero error to occur. While the profiler could decide that the divide by zero value is intentional it may turn out that the programmer only intended for a small range of values to be used or that the code itself is wrong. There is no way for the profiler to know this and may decide to not select that branch as primary even though it may be the primary path of execution through the program. This makes the use of a profiler a bit of a gamble that may generate accurate results but there is no easy way to ensure that this happens in all cases[2].

Compensation Code

Since region scheduling algorithms operate on more than one basic block there is the potential that instructions moved ahead of their original position could traverse the boundaries of a conditional branch. If the block is a *join* node then the block has more than one predecessor. If the block is a *split* node, then the block has more than one successor. Scheduling instructions across these boundaries can cause program correctness to be thrown out the window. To ensure that this does not happen the compiler must introduce *compensation code* to bring the program back into the realm of correctness. This compensation code consists of code require to ensure program correctness in a linear version of the program being scheduled[2]. There are four scheduling cases that where the generation of compensation code must be taken into account:

1. No Compensation

2. Join Compensation

3. Split Compensation

4. Join-Split Compensation

The first case, no compensation, occurs when the code motions that occur from scheduling do not change the relative order of operations with respect to joins and splits. This also covers the case where instructions are moved *above* a split point and become *speculative*

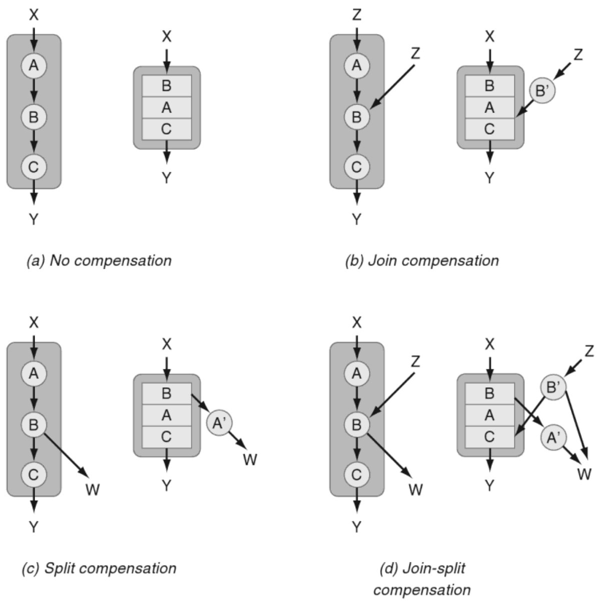

Figure 1.1: The Four Types of Compensation Code

instructions [2]. The second case, join compensation, occurs when an operation B moves above a join point A. When this happens, it is necessary to make a copy of operation B, known as B', and place it back below the join point to ensure data correctness[2]. The third case, split compensation, occurs when a split operation B move above a previous operation A. When this happens, it is up to the compiler to make a copy of A, called A', and put it back into the split path[2]. The final case, join-split compensation, occurs when either a join is moved above a split or a split is moved above a join. This is a pretty complicated case and the type of compensation code that is generated is equally complicated. There are actually several paths that the program can take through this region:

1. Control starting at the top of the block and leaving the bottom
2. Control entering the block through the join operation and leaving through the bottom
3. Control entering the top of the block and leaving through the other path in the split operation
4. Controls entering through the join operation and leaving through the split operation

Generating compensation code to ensure correctness in the face of these four conditions is not as complicated as it might seem. In reality, this is an extension of creating join and split compensation code with an extra branch to allow for the fourth condition of entering by join and exiting by split to be satisfied. This extra branch starts at the block copied in response to join compensation and links to the target of the split operation. This bypasses the entire block while maintaining program correctness[2].

These four types of compensation code must be taken into account and is probably the most error prone part of performing code scheduling. The generation of compensation code also becomes even more critical as the size of the scheduling area is increased through the use of region scheduling algorithms such as wavefront scheduling.

1.4 Wavefront Scheduling

Wavefront scheduling was developed to extract ILP for EPIC processors without needing to acquire branch statistics. It was developed by Intel for use in their proprietary compiler, icc, to improve performance on the Itanium architecture beyond what was feasible with basic block scheduling alone.

Getting Ready to Apply Wavefront Scheduling

Wavefront scheduling requires that a region be modified to support the changes that are going to be made when wavefront scheduling is applied. The first step in this process is the identification of scheduling regions that will be used in wavefront scheduling. These scheduling regions have to be acyclic and can contain so-called *side entrances*[3] and *side exits* [4].

While the exact algorithm icc uses to select regions is not known[5]. Intel has hinted at the fact that icc divides programs into program regions that consist of either the body of a function or a loop [4]. When dealing with loops the region selector turns each nested loop

[3]A side entrance is a block that has at least one predecessor within a given region and at least one predecessor that lies outside the scheduling region

[4]A side entrance is a block that has at least one successor within a given scheduling region and at least one successor that lies outside the scheduling region

[5]And probably never will become public knowledge

into it is own program region. This selection process turns every single part of the program into acyclic regions that satisfy the first condition of wavefront scheduling. Once a region is selected it is time to modify it by removing JS edges and inserting interface blocks.

Removing JS Edges A JS edge is an edge in a control flow graph that emanates from a *split* node and ends at a *join* node. Removal of a JS edge is done by adding an empty block, called a *JS block*, between the *split* node and the *join* node[4]. The insertion of JS blocks such that correctness is maintained is not always straightforward. This can occur when a join node has a label associated with it that multiple predecessors indirectly branch to. The solution to such a case involves inserting a fake JS block that the code scheduler is forbidden from writing into. This ensures that the block can be removed at the end of the scheduling process[4]. An example of JS Edge removal is provided in Figure 1.2 on the following page. Notice that in R' that blocks G , J , and K have been added to remove JS Edges present in R. Even though $F3$ is part of F and not R', the fact that E is a successor $F3$ requires the insertion of a JS node. The backedge from $F3$ to $F1$ is ignored because it is part of a loop.

Inserting Interface Blocks The second modification to the region comes in the form of inserting interface blocks. These interface blocks are empty blocks that are inserted before side entrances and exits to allow compensation code to be scheduled in[4]. Inserting interface blocks before actually performing wavefront scheduling also ensures that any compensation code that needs to be generated already has a block to insert the code into.

This modification is not necessary if the region selection algorithm the compiler uses disallows the creation of regions with side entrances and exits but in the event that it does the insertion of interface blocks is a necessary step. An example of interface block insertion is provided in Figure 1.2 on the next page. Blocks H and I are created to "pad" the exits to the given region in case it is necessary to insert compensation code to counteract the results of scheduling instructions into blocks ahead of the original position.

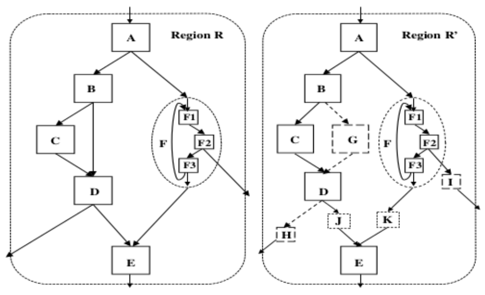

Figure 1.2: (a) While scheduling region R, code may be moved across nested region F but not into or out of F. (b) The region R with JS and *interface blocks* added.

Path Based Dependencies

With the region modified it is necessary to generate a list of all the distinct control flow paths through the region. This will allow wavefront scheduling to work in a far more efficient manner when it comes to selecting blocks to be contained on any given wavefront.

The number of distinct control paths through a given region is finite [4]. Yet the number of paths in a given region could increase exponentially with the number of blocks contained within the region. Thus it is really important to represent these paths as efficiently as possible. One such way involves the use of a bit vector where each bit represents a path in the region. When a bit vector is used for the purpose it becomes known as a *path vector*. These paths vectors not only define the number of paths but also allow subsets of the overall number of paths to be represented easily and efficiently [4]. Path vectors are also used to describe the content of each path.

The content of a path is created through the use of the *block path vector* function $BPV(n)$ which takes in a block n and returns a path vector denoting the paths through n. These block path vectors represent not only the content of each path but also control flow relationships such as dominance/post-dominance, control equivalence, and more[4]. These other relationships can be derived from applying bitwise operations to the block path vectors. For example, if one is given two blocks A and B then

- if $BPV(A) = BPV(B)$ then A and B are control equivalent
- if $BPV(B) \subset BPV(A)$ then A either dominates or post-dominates B

- if $BPV(A) \neq BPV(B)$ then A and B are control disjoint
- $\neg BPV(A)$ will yield all paths that A does not belong to

The process of enumerating all paths within a given scheduling region and generating the associated block path vectors is denoted by the following algorithm described in [4].

```
// Step 1: Calculate the number of paths NP($n$) through each node $n$
for each node $n$ in reverse topological order do
   NP($n$) = 0
   for each successor $s$ of $n$ do
      NP($n$) = NP($n$)+ NP($s$)
   end for
   if $n$ is an exit node then
      NP($n$)++
   end if
   BPV($n$) = $\{00\ldots00\}$
end for
// Step 2: Calculate the total number of paths through the region
numTotalPaths = 0
for each entry node $e$ do
   numTotalPaths = numTotalPaths + NP($e$)
end for
// Step 3: Compute block path vectors
k = 0
for each entry node $e$ do
   BPV($e$) = Bit vector comprising of $k$ low order zero bits followed by NP($e$) one bits,
   followed by remaining high order zero bits
   $k$ = $k$+ NP($e$)
end for
for each node $n$ in topological order do
   k = 0
   for each successor $s$ of $n$ do
      // Partition every pattern of NP($n$) contiguous ones in BPV($n$) into
      // $k$ zeros followed by NP($n$) ones followed by NP($n$)$-k-$ NP($s$) zeroes.
      // The result of this partitions is put into $pv$, and BPV($n$) is left unchanged.
      PV_Partition($pv$, BPV($n$), $k$, NP($n$), NP($s$), numTotalPaths)
      BPV($s$) = BPV($s$)$\cup pv$
      $k$ = $k$+NP($s$)
   end for
end for
```

Representing Data Dependencies

Once the control flow paths through the scheduling region have been constructed it is necessary to compute all of the data dependencies between different instructions throughout the entire region. The most efficient way to represent these data dependencies are through partitioning the information by instructions or registers.

When partitioning data dependencies through instructions the list of registers that the target instruction reads from and writes to are attached to the instruction. This process is repeated for every instruction within the scheduling region. While this method works great for basic block scheduling it has the issue of requiring the entire data dependency graph to be recomputed each time an instruction is scheduled. This is not only time consuming but also memory inefficient because entire sections of the graph have to be deleted and rebuilt continually. It also means that large sections of the graph will be recomputed even though it is not necessary.

The better method is to represent data dependencies by attaching a list of readers and writers to each register used by the region. This means that for every register v there are two sets $Readers(v)$ and $Writers(v)$ that contain the list of *instructions* that read from and write to v respectively[4]. The number of registers will usually be far smaller than the number of instructions in a given scheduling region. Representing the data dependencies this way also leads to the easy construction of so-called *def-use path vectors*. These path vectors represent the data dependence between any member w of $Writers(v)$ and a member r of $Readers(v)$ [4]. The path vector itself is defined to be the set of control flow paths along which the value v written by w is read by r [4]. These path vectors are computed using the function $DUPV(w, r, v)$ which consists of the following algorithm defined in [4].

```
for each $w$ in $Writers(v)$ do
  for each $r$ in $Readers(v)$ do
    if $w$ precedes $r$ then
      $DUPV(w, r, v) = BPV(\text{Block}(w)) \cap BPV(\text{Block}(r))$
    end if
  end for
end for
for each $w_1$ in $Writers(v)$ do
  for each $w_2$ in $Writers(v)$ do
    if $w_1$ precedes $w_2$ then
      for each $r$ in $Readers(v)$ do
        if $w_2$ precedes $r$ then
          $DUPV(w_1, r, v) = DUPV(w_1, r, v) - BPV(\text{Block}(w_2))$
        end if
      end for
```

```
        **end if**
      **end for**
    **end for**
```

Using *def-use path vectors* has the side effect of partitioning the data dependence graph into independent subgraphs based on registers. This means that, given registers x and v, $DUPV(w, r, v)$ will not be affected by any changes to the readers or writers of x where $x \neq v$. This has the side effect of allowing for efficient local updates of dependency information. For instance, if an instruction I is moved to another block in the scheduling region then each register x that I uses has to have all of its $DUPV(w, r, x)$ recomputed where w belongs to $Writers(x)$ and r belongs to $Readers(x)$. Using the algorithm above has a worst-case running time of $O(N_W^2 * N_R)$ where N_R is the number of readers and N_W is the number of writers for register x [4]. This makes the cost of updating the set of *def-use path vectors* for each register not very expensive and thus is far more viable for regions of varying sizes. However, it is possible to reduce the overhead of recomputing $DUPV$ for a given register by allowing the update process to be performed lazily.

When an instruction I is moved from one block to another, the *def-use path vectors* affected by this move are normally recomputed immediately. Instead, the *def-use path vectors* of each register used by I are marked as being *stale*. Any subsequent accesses to this information will cause it to then be recomputed. This prevents unnecessary updates from occurring as a result of multiple invalidations occurring between uses of the dependence information [4]. The use of *def-use path vectors* is a very efficient way to represent data dependencies without having to resort to multiple data structures. These path vectors are also very useful in the act of register renaming in the face of anti and output dependencies.

Renaming

When performing scheduling it is a really good idea to keep track of anti and output dependency information for instructions. An *anti-dependency* occurs when an instruction requires a value that is later updated. An *output dependency* occurs when the ordering of instructions will affect the final result of a register. These kinds of dependencies are usually called *name dependencies*[1]. These kinds of dependencies must be kept track of in some fashion to ensure program correctness. This information allows the scheduler to detect if moving an instruction would violate such a dependency and if renaming is required to eliminate the dependency. Keeping track of this information separately adds additional complexity onto the scheduler because it must be constantly updated and would slow down the process of compilation greatly. Instead, the *def-use path vectors* are used to acquire this information on a demand-driven basis [4].

For detection of anti and output dependencies to take place it is assumed that the instruction I involved in the code motion has already been moved to the new location. Then for each register v that is written by I, the path vector $DUPV(I, r, v)$ is recomputed for all $r \in Readers(v)$. If a change has occurred in $DUPV(I, r, v)$ then it is assumed that an anti or output dependency has been violated by the act of code motion. This violation requires renaming to occur to preserve program correctness. Violating anti and output dependencies creates flow dependencies and changes a flow dependency respectively. This simplifies the need to figure out if renaming is required by looking for flow dependencies generated by a prospective code motion[4].

Computing whether or not an instruction I requires renaming is performed by the following algorithm defined in [4].

```
for each result $v$ of instruction $I$ do
  $newDUPV = RecomputeDupv(I, v, targetBlock)$
  for each reader $r$ of $v$ do
    if $oldDUPV(I, r, v) \neq newDUPV(I, r, v)$ then
      return TRUE
    end if
  end for
end for
return FALSE
```

Once the act of renaming has occurred the last thing to do is to represent memory dependencies. This is different from register-based dependencies and consequently flow, anti, and output dependencies are computed and stored separately. This will create a memory dependency graph and it is necessary to assign each memory dependency edge with a probability based on the fact that memory references will interfere[4].

Once all of the register and memory dependence information is computed it is time to actually perform wavefront scheduling.

Understanding Wavefront Scheduling

Before performing wavefront scheduling it is necessary to know what a wavefront actually is. A wavefront is defined as a strongly independent cut set that partitions the scheduling region into three parts [4]:

- Blocks above the Wavefront
- Blocks on the wavefront
- Blocks below the wavefront

The fact that wavefronts are strongly independent implies that there is no control flow path in the acyclic region that flow through more than one block on the wavefront [4]. This means that all of the blocks on the wavefront collectively dominate all blocks below the wavefront and collectively post-dominate all the blocks above the wavefront. This ensures that compensation code can be inserted entirely into blocks on the wavefront when considering scheduling a candidate instruction I originally located in block B_k into a block B_w on the wavefront. The removal of JS edges ensures that every block in the region will be visited by at least one wavefront[4].

To ensure correctness while scheduling, the partitions implicitly created by wavefronts have different meanings. Any block above the wavefront is considered to be fully scheduled and thus code can not be added anymore. A block on the wavefront is not considered fully scheduled until it moves above the wavefront. This can cause some blocks to stay on the wavefront for an extended period of time. Finally, blocks below the wavefront are not scheduled into until those blocks become part of the wavefront[4].

Advancing the wavefront is done through the following algorithm found in [4].

```
*delNodes* = Set of *closed* nodes on the wavefront
**for** each node $n$ in *delNodes* **do**
  **if** $n$ has exact one successor $s$ **then**
    **for** each predecessor $p$ of $s$ **do**
      **if** $p \notin delNodes$ **then**
        $delNodes = delNodes - \{n\}$
        break
      **end if**
    **end for**
  end if
end for
for each node $n$ in $delNodes$ do
  $Wavefront = Wavefront - \{n\}$
  $Wavefront = Wavefront \cup SuccessorsOf(n)$
end for
```

This algorithm allows for some blocks to remain on the wavefront for an extended period of time. This means that potentially some instructions could be scheduled multiple times into the same block if the instructions are from a block that is not fully dominated by the block in question. Getting around this problem requires the introduction of deferred compensation.

Deferred Compensation

The control flow paths in the scheduling region R along which an instruction I needs to execute can be deduced from the original location of I and its data flow properties. This set of control flow paths can be represented as a path vector known as a *compensation path vector* denoted $CPV(I)$. When I is scheduled in a block K that does not fully dominate the block from which I originated[6], it is necessary to generate compensation copies of I to be scheduled elsewhere [1]. These needs are recorded in $CPV(I)$ instead of immediately generating compensation copies. This is done by scheduling an instruction I' which is a copy of I within block K. Then $CPV(I)$ is updated to reflect this by removing K from it. It is important to note that the generation of these instruction copies are deferred until they are actually scheduled. The instruction I left in the original block K represents all of the compensation copies that need to be generated as a side effect of the act of scheduling. The removal of the original instruction I from block K means that all copies of the instruction have been scheduled into every path requiring it. This process is known as *deferred compensation*. It is used to make the act of scheduling more flexible by not requiring instructions marked for scheduling to be immediately scheduled. This means that an instruction can be meant for scheduling but can be scheduled at a later time depending on the circumstances. *Deferred compensation* also prevents multiple copies of the same

[6]In other words $BPV(K)$ is not a superset of $CPV(I)$

instruction from being scheduled into the path multiple times. This is done through the introduction of a *lateness* constraint. This constraint says that an instruction that is subject to deferred compensation must be scheduled before the immediate successors of the block where the deferred instruction originated from becomes part of the set of blocks above the wavefront[4].

This introduces the concept of opening and closing blocks to allow the wavefront to advance over them. A block is considered *closed* when there is nothing more that can be scheduled into the block. This block can be re opened in response to another instruction being scheduled within another block on the wavefront. This allows blocks to flip between open and closed states in response to the changing conditions of scheduling[4].

1.5 LLVM

LLVM [7] is a compiler infrastructure that is written in C++. It is designed to be the code generation and optimization back-end for a wide variety of languages by making LLVM a library to be interfaced with. This has the side effect of giving programming languages that target LLVM immediate support for a wide variety of architectures and optimizations.

LLVM is very well documented and as such this section will be a bottom up explanation of different aspects of LLVM that are necessary to understanding the rest of the document. These aspects are values, users, instructions, basic blocks, constants, regions, loops, functions, the pass framework, and the module verifier.

One of the basic types of LLVM is the Value type. As the name implies, the value type represents some kind of value. Nearly every other type in LLVM is related to the Value type in someway. Direct descendants of Value include user and the basic block.

The user type is an extremely critical type within LLVM that represents a value that is used by another object. Both constants and instructions are direct descendants of the User class.

The instruction represents an operation to perform within the LLVM intermediate representation (or bitcode). There are over 50 different types instructions defined in LLVM IR

[7]LLVM used to stand for Low Level Virtual Machine. Now it does not stand for anything.

with each of them serving a separate purpose. However, each instruction contains a set of operands and a destination register to store the result into. Reading an LLVM instruction in its intermediate representation takes on the following form:

```
<destination> = <operation> <type> <operands>+
```

In this format there are several things that should be noted. First off, the operands of the instruction are users which means that they can be instructions, constants, or operators. The destination register is actually the unique name of the instruction itself. It is through this scheme that instructions are directly referenced. So the following instruction

```
%c = add i32 %a, %b
```

Is known by the name of its destination register c which has the side effect of any instruction that uses c will be using the instruction that computes c as well. This efficient representation requires that each register is unique to allow a one-to-one mapping to take place between registers and their corresponding instruction. This design also came about due to the use of single static assignment within LLVM bitcode. SSA stipulates that once a register has been set it can not be changed. This brings about the need to use phi nodes to converge values on multiple paths of execution to represent a single value. While this may sound a little confusing the following example should clear things up quite a bit.

```
int c;
if(a > b) {
   c = a * b;
} else {
   c = a / b;
}
```

While the value of c is set depending on which path is taken there is no way to have c be set in both the true and false paths of the value. This requires the creation of temporaries that are used by a phi node to populate the register c with the correct value based on the path taken. Applying phi nodes to the above example would transform it to look like this:

```
int c;
if(a > b) {
   c1 = a * b;
} else {
    c0 = a / b;
}
c = phi(c0, c1);
```

The phi operation would know which path was taken and thus select the correct value. In LLVM, these phi nodes are represented through the phi instruction which is best described by converting the above C code into a rough equivalent of it in bitcode:

```
%result = icmp sgt i32 %a, %b
br i1 %result, label %onTrue, label %onFalse

onTrue:
%c0 = mul i32, %a, %b
br label %finish

onFalse:
%c1 = sdiv i32, %a, %b
br label %finish

finish:
%c = phi i32 [ %c0, %onTrue ] , [ %c1, %onFalse ]
; Whatever else is necessary
```

Reading the phi instruction takes the following form

- if onTrue was taken then select c0

- if onFalse was taken then select c1

While this example only has two paths, the phi node must have a value for each path that the parent basic block is a part of. It is also important to note that phi nodes are not directly translated into machine code. Instead, they are used to describe the flow of computation for register allocation. The use of SSA removes so called anti-dependencies that come about from allowing registers to be modified after their first definition. These anti-dependencies make it harder to apply certain optimization such as common sub-expression elimination and basic block scheduling. The rest of the instructions provided by LLVM bitcode are extremely straight forward and will not be discussed in this section.

A constant in LLVM represents not only constant values such as numbers but anything that does not change during the course of the execution. This includes arrays, addresses, structs, global values, expressions, and much more. While the majority of these types makes sense there are a few that are a little strange and must be discussed in detail. They are constant arrays, structs, and expressions. A constant array or struct is not an array or struct that does not change. Instead they are an array or struct that was defined in a constant aspect. In C, this would be an array of fixed size or a structure that is directly allocated. The addresses of these kinds of structures will not change and are fixed in memory for their lifetime. A constant expression is a way of representing an instruction that will have a constant result. The most frequent use of a constant expression is a constant getelementptr instruction that will always compute the same offset in memory.

Since a constant is a user, it can be used as an operand in an instruction. It is totally plausible to see an instruction like

```
%result = add i32 %a, i32 9
```

which adds nine to register a. With this understanding of constants it is now possible to talk about basic blocks.

A basic block in LLVM follows the definition of a basic block described earlier. Obviously, this means that a basic block is aware of several things including its immediate set of successors and predecessors; and the instructions contained within it. The basic block has few interesting requirements imposed upon it by LLVM. The first requirement is that phi nodes are the first instructions in the block. This is to allow any values that need to be converged to be ready before the code in the block can be executed. The second requirement is that the block always ends in a terminator instruction. A terminator instruction is a subtype of instruction that includes branches, switch statements, returns, and many more. The requirement of having each block end in a terminator also extends to the requirement that terminator instructions are only allowed to be the last instruction in a given block. This makes it really easy to create basic blocks and keep things clean and consistent. However, basic blocks are just a small part of the structures provided by LLVM. There are also regions and loops.

A region, in LLVM, is a set of blocks that has a single entrance and a single exit. There are a few aspects to regions that may be confusing if the behavior is not defined ahead of time. First off, the exit block of a region is not owned by the region itself. This is to allow regions to flow directly into other regions. The second aspect is that a region is that the set of blocks that make up the region may not be directly owned by the region. It is possible that the region also contains sub-regions as well. When this is the case, the entrance to the sub-region may not be owned by that subregion but one of its children. The third issue with regions is that they have zero awareness of any loops that may be contained in the target region or if the region just happens to represent a loop. Both of these cases can be extremely hazardous if not properly handled.

Loops in LLVM are defined as single entry natural loops as defined in [1]. Loops contain similar issues to regions in that they are defined without taking regions into account. The great strength of the loop is that it can have multiple exits with each of those exit blocks not being owned by the loop. The only downside to the way LLVM creates loops is that they have to be natural loops. This means that cycles, which are loops that have more than once entrance, will not be identified. Generally speaking this is not an issue because the majority of loops will fall under the guise of being a natural loop.

A function in LLVM is an independent set of blocks that takes in arguments and could easily be described as a super set of a region that can have a single entrance but multiple exits. Functions within LLVM are interesting because they are self contained with respect to control flow. LLVM performs checks to see if a function has side effects, writes to memory, and reads from memory. This is useful for not only functional programming languages but for the purpose of scheduling as well. A function without side effects allows those instructions that are independent of the result of the function's call to be scheduled ahead of it.

With all of the information described it is now possible to talk about the pass framework that LLVM uses to define and execute optimizations. This pass framework is defined in an object oriented fashion to make it really easy to use. There are a series of passes with the most common ones operating on basic blocks, loops, regions, or functions. There are other passes but these four are the main ones that most programmers will extend off of in some fashion or another. The power of the pass framework is not only in the ability to create optimizations but also define dependencies as well. This allows programmers to create execution pipelines allowing data to be passed from one optimization to the next. Passes are also interesting in that they are not allowed to maintain state. This is stipulated to reduce the memory footprint of LLVM while allowing it to potentially take advantage of threads in the future.

The module verifier is a compiler pass that analyzes an LLVM module for issues such as using a register before it is defined, having unlinked instructions use a value, creating a phi node that does not converge all paths, and much much more. In LLVM terms, a module is container class for any aspect of the LLVM intermediate representation. In the confines of this paper the module verifier runs after each function has been wavefront scheduled. If it detects an issue then it will crash the optimization with a meaningful message and a stack trace that can sometimes be used in GDB to diagnose what the problem is. The module verifier is a wonderful pass that makes implementing compiler passes a far more enjoyable experience.

Now that the description of LLVM is out of the way, it is time to acclimate the reader in the CLIPS programming language and expert systems.

1.6 An Introduction to the CLIPS Expert System

CLIPS stands for C Language Integrated Production System and is a public domain expert system creation tool developed at NASA in the 1980s. It is designed to model *rules of thumb* that a human expert would use to solve a problem. Unlike other programming languages, CLIPS operates on knowledge instead of data. Knowledge is different from data in that it is data that has been given explicit meaning. The conversion from data to knowledge is known as *knowledge construction.* Within CLIPS knowledge is given a generic framework to exist in that take the form of facts, instances, and templates. A fact is a collection symbols, numbers, or strings that have special meaning. Defining a fact is easy and is performed by *asserting* it. For example:

```
 (assert (This is a fact))
```

This causes a fact to be created that contains the symbols This, is, a, and fact. Unless specified, everything in CLIPS is a symbol. The exception to this rule are numbers, strings, and variables [5]. It is important to note that, unless disabled, every single fact must be unique. This means that attempts to assert a fact that is exactly the same as a fact that already exists will result in nothing happening[8].

Operating on this knowledge occurs through *rules* which describe actions to be taken when a given condition is met. It is important to note that rules will not fire until the user explicitly says to do so through the use of the run command. The following code shows the declaration of a simple rule.

```
(defrule BoastAboutFact
 (This is a fact)
 =>
 (printout t "HAHAHAHA! I have a fact!" crlf))
```

A rule definition is made up of a rule-header, a set of conditions made up of *pattern-entities*, and a body of code, following the =>, which is invoked upon successful match of *all* pattern-entities. A pattern-entity is either a fact or an instance of a user-defined class[5]. Each pattern-entity is described by being surrounded in its own set of parenthesis. In the example above the pattern-entity matching against the fact previously asserted denotes that the rule is looking for a fact that is made up of the symbols This, is, a, and fact in that order. Once all pattern-entities meeting the criteria for matching the target rule have been satisfied then the rule is retracted from service. This can be undone if there are new facts asserted that cause the match to occur again. The matching behavior of rules cause the firing of them to be *unordered*. This requires the programmer to think in terms of actions more than process. With that out of the way let us talk about what this rule actually does. It requires that there is a fact of the form (This is a fact) and when that is the case it will printout the above message. While the message makes sense the other arguments to printout may confuse some. The use of the letter *t* is meant to tell CLIPS to print the message out to the standard output stream. The symbol crlf represents a newline character. So the message will printout out the message *HAHAHAHA! I have a fact* followed by a

[8]This feature is exploited throughout the wavefront scheduling implementation.

newline to the standard output stream.

However, the above example is just scratching the surface with respect to what facts and rules are capable of. Let us define some more complicated facts to demonstrate some of this power.

```
(assert (Age 20))
(assert (Age 21))
(assert (Age 22))
(assert (Age UNKNOWN))
```

There are now several facts describing the age of something in different terms. It is possible to write one rule to match against all of the different facts through the use of variables. Unlike other programming languages, variables start with a ? in CLIPS. Variables can be defined on the conditional side of a rule as shown in the following rule

```
(defrule PrintAge
 (Age ?age)
 =>
 (printout t "Found a thing aged " ?age crlf))
```

The pattern-entity describing the age fact is generic and will accept any fact that is of the form (Age *Element*). The rule will automatically bind the value of the target element to the variable ?age. This is an extremely powerful tool that opens quite a number of possibilities. One such possibility is the ability to generically retract facts that have been matched to a given rule. So what is the output of this rule? Well this rule will fire several times and will have the following results

Fact Selected	Value of ?age	Result
(Age 20)	20	prints *Found a thing aged 20* to the console
(Age 21)	21	prints *Found a thing aged 21* to the console
(Age 22)	22	prints *Found a thing aged 22* to the console
(Age UNKNOWN)	UNKNOWN	prints *Found a thing aged UNKNOWN* to the console

Notice that the variable ?age is passed directly into printout and the *value* stored in ?age is printed out. This is allowed because printout takes in a variable number of elements that are printed out in the order specified.

Let us modify the above rule to remove the fact after it has been matched. The process of removal is known in CLIPS as *retracting* the fact.

```
(defrule PrintAge
 ?fct <- (Age ?age)
 =>
 (retract ?fct)
 (printout t "Found a thing aged " ?age crlf))
```

It is quite obvious that things have changed as the *address* of the target fact is bound, through the use of the $<-$ operator, to the variable ?fct[5]. The use of variables in pattern matching also streamlines the ability to match against facts or instances that contain the same value defined in the target variable. This rule matches against an age fact with the address of the fact being bound to the variable ?fct. If the match is successful then the fact is retracted and the message *Found a thing aged* ?age is printed to the console.

Once again, the PrintAge rule is going to be redefined to match against a fact that describes a person of the given age.

```
(defrule PrintAge
 (Age ?age)
 (Person ?name is aged ?age)
 =>
 (printout t "Found a person named " ?name " aged " ?age crlf))
```

The variable ?age found in the two patterns refers to the same value and as such is the same as checking to see if the value of age in the second pattern is equal to ?age. It is important to note that this rule will be fired the same number of times as there are Person facts with age ?age. This rule will be fired if there is a "age" fact with a variable ?age that is the same as the age defined in a given "person" fact. The name of this person is bound to ?name. If this match is successful then the message "Found a person named ?name aged ?age" is printed out to the console with a newline.

However, let us say that it is necessary to find people with ages *greater* than a given age fact. This can be expressed in two different ways.

```
(defrule PrintAge
 (Age ?a1)
 (Person ?name is aged ?a0&:(> ?a0 ?a1))
 =>
 (printout t "Found a person named " ?name " who is older than " ?a1 crlf))
```

In this example the & symbol is used to describe that there is a value ?a0 at the specified position *and* ?a0 is greater than ?a1. In this rule, the match will only happen if the value ?a0 is greater than ?a1 from the "age" match. If this match occurs then the message "Found a person named ?name who is older than ?a1" will be printed to the screen. Of course, the variables ?name and ?a1 will be replaced with the value each contain before actually printing out to the console.

The & operator can also be replaced with the test operator to do basically the same thing.

```
(defrule PrintAge
 (Age ?a1)
 (Person ?name is aged ?a0)
 (test (> ?a0 ?a1))
 =>
 (printout t "Found a person named " ?name " who is older than " ?a1 crlf))
```

The test operator is different from the & operator in that it will not fire unless all preceding pattern-entities have been matched first. The test operator is also incapable of being the first pattern entity of a given rule[6]. The exact reason for this behavior is beyond the scope of this paper. This rule does the exact same thing as the version before it.

Everything up to this point has been around facts that consist of a known size. Handling facts that may consist of an unknown size requires the use of a *multifield*. A multifield is a collection of zero or more elements and variables describing them are denoted by having a $ in front of the ?. The multifield is a very powerful tool in CLIPS and allows for very complicated yet elegant pattern matching to occur in rules. It also allows rules to "iterate" over an unknown number of elements. This "iteration" is described by the following example

```
(assert (Elements: A B C D))
(defrule PrintoutElements
 (Elements: $? ?current $?)
 =>
 (printout t "Looking at " ?current crlf))
```

This rule "iterates" through the target fact by matching the fact in as many ways as possible. The lack of variable names on the $? just means that the value described there is meant solely for matching, nothing more. Printout elements will fire four times but not in the order originally believed. This is due to the unordered nature of the expert system.

Generally speaking the order of evaluation of the above rule will be something like the following table:

First Multislot	Current Value (?current)	Second Multislot
(A B C)	D	()
(A B)	C	(D)
(A)	B	(C D)
()	A	(B C D)

Obviously this rule prints out the element that it is currently ”looking at”. An element in the given fact is selected and is printed out to the screen as described in the rule.

It is possible to have far more complicated multifield matches than the rule described above. Multifields need to be used with care as really complicated matching patterns will generate a significant amount of overhead due to the fact that *copies* of the different matched sections have to be made. This can cause memory usage to swell and slow the program down greatly if the multifield match is really complex.

CLIPS also has the ability to describe knowledge in terms of objects. An object is defined by zero or more slots and is able to inherit from multiple sources. As an example let us define the following class

```
(defclass Container (is-a USER)
 (slot Parent (type SYMBOL INSTANCE_ADDRESS))
 (slot IsASet (type SYMBOL) (allowed-values FALSE TRUE))
 (multislot Elements))
```

While this class declaration looks different than those seen in Java and C# the idea is the exact same. Each slot describes a field where data can be stored. The type modifier describes the values that can be legally stored in the slot. The allowed values field describes the exact values that can be legally stored in the target slot. In the case of the second slot, only true and false values are allowed. The final slot is a multislot which is class version of the multifield. Defining and using objects are fairly easy.

```
;Build an instance of the class
(make-instance [A] of Container (Elements A B C D E F A B C D E F))
;now let's "setify it"
(defrule SetifyContainer
?obj <- (object (is-a Container) (IsASet FALSE)
                (Elements $?before ?curr $?middle ?curr $?rest))
=>
;this will continue to run until all elements are unique
(send ?obj put-Elements (create$ $?before ?curr $?middle $?rest)))
```

CLIPS uses a message passing scheme to modify objects. This is done through the use of the send command which takes in the target object's instance name or address. In the rule above the instance address is bound to ?obj. Instance names can be explicitly passed through the use of the square brackets which denote the symbol inside of the brackets is the name of an instance. The third argument of the send function is the message to send. In this case it is the put-Elements command which is implicitly defined by CLIPS during the definition of the class. Unless explicitly stated, each slot in a given class has a get and put message associated with it. The only caveat with sending messages to objects is that they are very slow due to the overhead associated with checking to see if the class has a corresponding message handler. Thus it is a good idea to minimize the number of messages sent. With that being said there are times when this is unavoidable.

This rule will fire on the case where the same element is found twice within the Contents multislot of a given container object. The second instance of the duplicate will be removed from Contents of the target object by replacing it with a copy of the list without the duplicate element.

In the above example there is no way to safely denote that the list is now a set. As it stands right now there is no way to delay the firing of a rule so that the IsASet slot can be set to TRUE after *all* duplicate elements have been removed.

Solving this issue requires the use of the salience command which explicitly sets the priority of a given rule. This priority is expressed as a number between -10000 and 10000. By default, all rules have a default salience value of zero[5]. Higher salience values equate to the rule in question being fired earlier than those with lower salience. Let us add another rule that will denote the target Container as being as set after the container has been converted into a set.

```
(defrule MarkContainerAsBeingASet
 (declare (salience -1)) ; this _has_ to be the first command
 ?obj <- (object (is-a Container) (IsASet FALSE))
=>
 (send ?obj put-IsASet TRUE))
```

With this rule in place any Container object that has been turned into a set will have the corresponding field denoting that as well.

Earlier it was mentioned that knowledge is expressed through the use of fact, instances and templates. While facts and instances have been described, templates have not. A template is best described as an unordered fact. It has the slots of an instance, can only be modified by being retracted, and is asserted in the same way as a fact. The following is an example of defining, using, and modifying a template.

```
; For the sake of familiarity let us use the Container class
(deftemplate Container
 (slot Parent (type SYMBOL INSTANCE_ADDRESS))
 (slot IsASet (type SYMBOL) (allowed-values FALSE TRUE))
 (multislot Elements))

; Notice that it is similar to an object in creation
; The default modifier can be provided so that specific fields to not have to
; be provided.
(assert (Container (Parent nil) (IsASet FALSE) (Elements A B A C D B)))

(defrule MakeContainerASet
 ?fct <- (Container (IsASet FALSE) (Elements $?a ?b $?c ?b $?d))
 =>
 (modify ?fct (Elements $?a ?b $?c $?d)))

(defrule MarkContainerAsASet
 (declare (salience -1))
 ?fct <- (Container (IsASet FALSE))
 =>
 (modify ?fct (IsASet TRUE)))
```

This code defines a Container template and then creates one. When the expert system is run the MakeContainerASet rule will fire as many times as necessary to delete all duplicate elements. Once that has been completed the MarkContainerAsASet rule will fire and modify the template instance such that it is marked as being a set.

It is important to note that when a template is modified that it is retracted and a new version of the template is asserted. This is different from an object where memory is modified in place. Facts, instances, and templates each have a role to play when writing a program. Objects are useful when there is a lot of similarity between different structures and it is useful to be able to match against super types. Templates are nice when a fact is desired but a fixed order is not desired. Facts are generally useful because of their flexible nature.

With this introduction to understanding how CLIPS works it is now possible to understand how wavefront scheduling was implemented using CLIPS as the majority of this paper details how the implementation works.

Chapter 2

LLVM Implementation

2.1 Introduction

Implementing wavefront scheduling required the use of a compiler infrastructure that was easy to add on to, has well documented internals, and was fast. LLVM met all of these conditions quite well and never really *got in the way* of implementing wavefront scheduling. It was decided to take a *hybrid* approach by integrating CLIPS into LLVM. Implementing wavefront scheduling as an expert system was the easiest way to represent the algorithm and make it maintainable. The features provided by CLIPS that were the most attractive include:

1. The language is interpreted
2. Rule based
3. Application of rules is unordered
4. Automatic memory management
5. Automatic management of facts and instances
6. Use of facts to drive changes
7. Extensive integration guide
8. Well documented

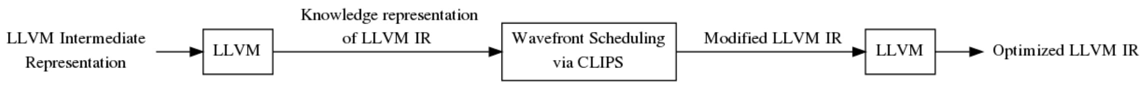
LLVM Intermediate Representation
LLVM
Knowledge representation of LLVM IR
Wavefront Scheduling via CLIPS
Modified LLVM IR
LLVM
Optimized LLVM IR

Figure 2.1: A high level overview of how wavefront scheduling interacts with LLVM.

These features make it very simple to just implement the algorithm and not have to worry about the order of application. While the majority of the wavefront scheduling implementation was written in CLIPS there were still critical aspects written in C++. This includes the removal of JS edges, converting the LLVM representation of the target function into knowledge usable by CLIPS, the creation of functions that would allow CLIPS make changes within LLVM, and the pass that actually invokes wavefront scheduling. A high level view of this system architecture is defined in Figure 2.1 while a detailed version is found in Figure 2.2 on page 53.

2.2 Removing Join Split Edges

As stated earlier, wavefront scheduling requires the removal of join split edges which is an edge that directly connects a split node to a join node. Removing such an edge requires the insertion of a block onto that edge that breaks up the direct connection. These nodes are critical to wavefront scheduling because they provide the means to ensure that scheduling occurs on all paths for a given instruction. Without these nodes it is entirely possible that a given instruction may remain in the original block and reduce performance by requiring the scheduled instruction to be computed twice so that correctness is maintained.

Originally, this action was performed within CLIPS but it was very difficult to properly update the control flow graph (CFG) within LLVM and CLIPS. Thus it was decided to move this action into LLVM as a separate pass that is executed before the application of wavefront scheduling. This turned out to be a very smart move because LLVM contains a function called *SplitCriticalEdge* which performs JS edge removal. Thus it was really simple to just iterate through all blocks in a given function and remove JS edge nodes. The only exception to this simplicity is that blocks that contain an indirect branch statement are not valid targets for JS edge removal. This limitation is enforced by LLVM itself because the actual target that a indirect branch statement jumps to is usually not known at compile time.

The only thing that this function pass does not do is pad entrance and exit blocks. It was not deemed necessary due to the fact that an LLVM region is a single entrance affair where the exit blocks of a given region are not owned by the region itself. However, during testing it was found that there are several cases where entrance and exit block padding is necessary to allow blocks that converge several distinct paths of executions to be valid targets for region scheduling. This will be implemented in a future version of this pass.

The overall running time of this pass is $O(n^2)$ where n is the number of basic blocks in the function. This worst case running time is computed from the fact that each block of the function is evaluated and assuming that it does not have a indirect branch then the list of successors of that block is iterated through. This list of successors has an upper bound of the number of blocks within the function due to the fact that functions are self contained within LLVM. It is important to note that this does not take running time of *SplitCriticalEdge* into account as the objective of the running time analysis of the pass was to determine how much overhead the pass itself has not any external functions it calls.

Once JS edge removal is complete, control is passed to the wavefront scheduling pass.

2.3 Integrating CLIPS Into LLVM: The Knowledge Construction Engine

The wavefront scheduling implementation takes a hybrid approach that consisted of code written in C++ and CLIPS. The reason behind this choice has mainly to do with the fact that CLIPS already has a garbage collector, built-in pattern matching, and automatic management of all facts and instances. This allowed the implementation to be more about wavefront scheduling and less about the mechanisms that ensure that wavefront scheduling is being applied correctly.

This choice brought about some very interesting side effects that highlight the biggest disadvantage of using an expert system: *knowledge engineering.* Thus the first step to using CLIPS to implement wavefront scheduling was actually providing CLIPS with the knowledge it needed before a single rule could even be fired. Thus it was necessary to build a *knowledge construction engine* that converts an LLVM function into a form that CLIPS operate on.

This knowledge construction engine, or KCE, is quite simple but the construction and subsequent optimization of it yielded some interesting and unexpected results. The main components of the KCE are the FunctionNamer class, the CLIPSObjectBuilder hierarchy and a system of object routing.

2.3.1 Naming Anonymous Objects

One of the biggest jobs of the KCE is to provide LLVM objects with unique names. This is necessary because building up the CLIPS representation of these objects requires a name. If the LLVM object is not of void type then these names can be applied to it as well. This task is left up to the FunctionNamer class which keeps track of all things having to do with names for a given function. It is not a singleton in any regards but there should only be one instance of the object per function. The FunctionNamer class is capable of

- Generating unique instruction names
- Generating unique loop names
- Generating unique region names
- Generating unique basic block names
- Generating unique generic names
- Binding a pointer address to a given name

Each type has a 64-bit number associated with it that is incremented each time a new name is required. This has the side effect of allowing generation statistics to be kept track of by allowing the current value of the counter to be exposed in a read-only fashion. Each name type is differentiated by the character that is appended to the front of the number to construct the name.

- Instructions start with *i*.

- Loops start with *l*.
- Regions start with *r*.
- Basic blocks start with *b*.
- Generic symbols start with *g*.

There are two ways that a name is generated, using a char pointer or through the use of a LLVM raw string output stream which is a wrapper over a standard C++ string that will concatenate not only the default set of C++ types but LLVM specific ones as well. In the current implementation, the char pointer method is used more often as it is easier to use with the object building classes.

Later on it was discovered that LLVM reduces its memory footprint by making a single instance of each unique object. If there are three objects that all are of type Integer. Then the Type object associated with those three objects will all be pointers to the same Type instance. This provided the inspiration for retrofitting the FunctionNamer class to keep track of what name went to which object. This is done by using an LLVM provided data structure called a DenseMap which is designed to store a large number of different keys while allowing quick and efficient access time to the elements stored within it. The map uses the pointer address of the given object as the key and the name as the value. This information is used during the routing process to determine if it is necessary to actually construct a CLIPS object for the given object. If the pointer is registered then the corresponding name is returned.

It is important to note that this process is not done when a name is generated, instead the use of this functionality is left open to the CLIPSObjectBuilder and its hierarchy.

The FunctionNamer is a very critical component of KCE because it governs not only naming but also determines if an object really needs to be constructed. Without it, the KCE would be much larger and far harder to extend.

2.3.2 The Object Building Hierarchy

Converting LLVM object into CLIPS knowledge really consists of building a CLIPS object with the same fields as the LLVM one. Although CLIPS provides functionality to build objects in a piecemeal fashion, this was found to be way too slow within the confines of a compiler optimization as even a relatively simple function could take a second or more to build! It was found that using the MakeInstance[7] function which takes in a string of the form (Name of Type Fields*) was significantly faster because the interaction with CLIPS would be reduced solely to constructing the object and nothing more. It is up to the KCE to build this string and pass it to CLIPS.

This string construction is abstracted away through the use of the CLIPSObjectBuilder class and its children. These classes are tasked with building the class string, passing it to CLIPS, and ensure that the formatting is correct. This not only removes the tedium and error prone nature of manual string concatenation but also divides the process into three phases: pre-build, build, and creation.

The pre-build phase of a CLIPSObjectBuilder instance is tasked with starting the construction of the instance string by setting the name and type of the CLIPS representation of the target LLVM instance. When a CLIPSObjectBuilder is created the object is automatically in this phase. Switching to the next phase requires the use of the open method which will construct the beginning of the instance string. Once open has been called it is possible to start populating the fields of the object with values. The build phase is responsible for this by allowing the programmer to pass in the actual LLVM instance of the target object to a method called addFields which will set all of the proper fields in the corresponding instance string. This is done in a recursive fashion to cut down on the amount of duplicated code. Once the object has been "transcribed" into the instance string the only left to do is to call the close method. This will cause a shift over into the creation state. The objective of the creation stage is to call the MakeInstance function which will convert the instance string into a corresponding CLIPS object. This creation is done through a call to instance method convertToKnowledge. Once this method has been called the act of knowledge construction for the corresponding object has been completed.

It is important to note that any sub objects of the target object will be fully constructed in the build phase. This is simplified through the use of the routing system of the KCE which will be discussed in further detail later on.

The object hierarchy of the KCE is modeled after the LLVM object hierarchy and as such are quite similar.

There are over fifty different sub types to the object hierarchy which would make it difficult to use in the cases where objects contain elements of a super type such as User. The problem is that the exact type of the given object needs to be known to allow accurate knowledge construction to occur. Solving this problem required the introduction of a routing system that is used by the hierarchy to dispatch an unknown type for construction without needing to know what it is.

2.3.3 Object Routing System

Normally, it is almost impossible to figure out the exact type of an object in C++ due to the fact it lacks the reflection facilities required to do something like that. Fortunately, the object hierarchy in LLVM is fitted with functionality to allow the exact type of an object to be determined at runtime. This functionality takes the form of the static functions isa and dyn_cast. The isa function performs a test to see if the given object is of a specified type and returns a boolean value signifying if it the case or not. The dyn_cast command takes the isa command a step further and attempts to cast the object into the specified type. If the object is of the specified type then the casted object is returned otherwise zero is returned. This allows an if statement to check to see if the object is null or not. In C++ the check takes the form of seeing if the resultant value is the null pointer or not.

The router system takes advantage of these functions by providing functions that take objects of a generic subtype and uses the dyn_cast command to determine what kind of object it is. This search is broken up into different generic super types including: Value, User, Instruction, BasicBlock, Loop, Region, Argument, Constant, Type, and Operator. These types are bound to a different routing function that takes in the object instance, the parent of the target object, and the FunctionNamer object of the current object. The function will then determine what the actual type of the object is, create the appropriate object builder for that type, build the CLIPS representation of the target object, and returns the CLIPS name of the new created object. If the given object has already been created then the symbolic name is acquired from the provided FunctionNamer class and returned.

Using the object router system is quite easy and only requires a call to Route with the target object, its parent, and the FunctionNamer object associated with the current function. All of this information is known by any object builder instance which makes it easy to use.

The object router system is designed to be as modular as possible. Adding a new subtype requires the creation of an associated object builder class and entries within the routing function that the subtype is a child of. For instance, if a new type class had to be added then an entry for this new type class would be added to the route function that takes in an object of type Type. Then next step would be to recompile the module and the object router would automatically be ready to handle the new subtype.

2.4 Allowing CLIPS to interact with LLVM

Allowing CLIPS to interact with LLVM was a topic that required some ingenuity to solve adequately. Since LLVM is written in C++ it was not straight forward how CLIPS would direct LLVM to make changes during the act of wavefront scheduling.

The solution to this problem took the form of passing the pointer address of the underlying LLVM object to its CLIPS counterpart. This not only prevented the need for extensive book keeping by LLVM but also reduced the overhead of having CLIPS communicate with LLVM by only needing to pass pointers of objects that need to be changed in some way. The functions that allow CLIPS to perform these changes to LLVM are written in C++ and then added to CLIPS through the methods described in [7]. While the KCE generates a considerable number of objects, there are only seven functions necessary to allow CLIPS to drive changes in LLVM. These functions are CloneInstruction, UnlinkAndMoveInstructionBefore, UnlinkAndDeleteInstruction, CreatePhiNode, ReplaceUseOf, ReplaceAllUsesOf, and ScheduleInstructions. These are the names of corresponding C++ functions that is called indirectly by CLIPS through an aliased name that will be noted with the description of each function below.

CloneInstruction does exactly what its name implies. It makes a clone of a given instruction within LLVM. This is done through a call to clone which returns a new instance of the given instruction. The pointer to this instruction is returned back into CLIPS which becomes the value stored in the Pointer field within the cloned instance. This function also takes in a new name for the clone. If it turns out that the instruction is unable to be named then the name is not set. Within CLIPS, this function is known as llvm-clone-instruction.

UnlinkAndMoveInstructionBefore calls takes a target instruction removes it from its old parent and inserts it before another instruction provided to the function. If the action was successful then true is returned, otherwise it is false. This function is known as llvm-unlink-and-move-instruction-before within CLIPS.

UnlinkAndDeleteInstruction is used to erase a target instruction from its parent. It calls the instruction's method eraseFromParent which deletes the instruction after removing it from the parent. This function is known as llvm-unlink-and-delete-instruction within CLIPS.

CreatePhiNode does exactly what it sounds like, creates a phi node within LLVM. The only catch to this simplicity is that it requires five arguments. The first argument is the new name of the phi node. The second argument is a pointer to the type of data this phi node works with (double, float, int, etc). The third argument is the number of paths this phi node represents. The fourth argument represents the pointer address of the instruction to add this new phi node before. The fifth argument contains the list of elements that make up the phi node. The count provided is half the number of elements contained within the multifield that makes up the list of elements to add. This function takes these arguments, constructs a new phi node and inserts it before the instruction provided. The address of this new phi node is returned into CLIPS as the pointer of the CLIPS representation of the newly created phi node. This function is known as llvm-make-phi-node within CLIPS.

ReplaceUsesOf replaces a given instruction with another instruction within LLVM for a given set of instructions. This function is used to provide fine grain control over replacing uses that would be lost if a straight call to replaceAllUsesWith were made. This function iterates through the provided list and call replaceUsesOfWith manually on each one. This function is known as llvm-replace-uses within CLIPS.

ReplaceAllUsesOf is a CLIPS wrapper over a call to replaceAllUsesWith for a given instruction. This function takes in two arguments, the pointer to the original target and the pointer to the replacement instruction. It then uses the replaceAllUsesWith method of the first instruction to replace it with the second instruction. This has the effect of replacing all uses of a given instruction with another within LLVM. This function is known as llvm-replace-all-uses within CLIPS.

The final functions ScheduleInstructions takes a list of pointers and "schedules" them by moving them to the bottom of the same block. It is up to clips to provide the ordering to this list of pointers however the results will always be correct and once all of the instructions have been moved then the basic block will have been scheduled within LLVM. This function is known as llvm-schedule-block within CLIPS.

It is really important to note that these functions are designed to be as efficient as possible. Breaking out of CLIPS into LLVM is extremely expensive if done continually. If a new function is to be added it should be designed to operate on a set of data items instead of just one. Generally speaking it is most efficient to only *inform* LLVM of changes when necessary.

2.5 The Wavefront Scheduling Function Pass

The wavefront scheduling pass itself is actually very simple. It is less than sixty lines of code and acts as what can only be described as the initiator of both the KCE and CLIPS itself. The pass is actually a function pass instead of a region pass due to the expense involved with invoking the KCE. Making the pass operate on a function instead also makes it extremely flexible with respect to future improvements or changes.

In its current form the wavefront scheduling pass is broken up into several parts including:

1. Initializing the CLIPS environment and all code associated with the act of wavefront scheduling.

2. Checking to see if the function is a viable candidate for wavefront scheduling.
3. Providing a name to all unnamed LLVM objects.
4. Converting the LLVM representation of the given function and its contents to a CLIPS knowledge based representation
5. Invoking CLIPS

When an instance of WavefrontSchedulingPass is created two major things happen: the CLIPS environment is initialized and all of the source code associated with wavefront scheduling is loaded into it. Once this is complete the pass is ready to accept functions to apply wavefront scheduling to. When a function is passed to the WavefrontSchedulingPass the first thing that the pass does is check to see if the function has more than one block. If the function only has one block then it is not considered for wavefront scheduling due to the fact that it would be a complete waste of time to do so as there is nothing to schedule. When the provided LLVM function object only has one block the application of wavefront scheduling is skipped and control is passed back to LLVM.

However, if there is more than one block in given function then the environment is reset which clears out any facts and instances from a previous run. This reset does not erase any rules, functions, or class definitions. Next, the function object is modified such that all basic blocks and instructions are given names by the KCE. Then the loops and regions contained within the function object are acquired and converted into a corresponding CLIPS representation through the use of the KCE. This conversion handles not only regions and loops but also the objects that are a part of them as well. This includes instructions, basic blocks, nested regions, nested loops, and constants. The order of conversion starts with the loops of the function and then moves over to regions to convert the rest of the function object.

LLVM has no facilities for returning loops and regions in the same structure. As such it is necessary to merge these structures together in CLIPS because it is not only easier but far less error prone. However, it is up to LLVM to inform CLIPS of the changes that need to be made and as such it is a function of the region object builder in the KCE to assert the corresponding facts to start this process.

Once all of the regions have been routed through the KCE the entire function will have been converted for CLIPS use. The final thing to do is to call the Run[7] function provided by CLIPS to start the process of wavefront scheduling. Once wavefront scheduling has been completed in CLIPS, control is passed back to LLVM which notes that changes have been made to the intermediate representation. These changes require reanalyzing the corresponding function object before further optimizations can be applied.

The objective of the wavefront scheduling pass is to convert the provided function object to a corresponding CLIPS version and then apply wavefront scheduling to this CLIPS representation. This made the pass not only simple but easy to debug as well. At the time of writing of this document, the WavefrontSchedulingPass object is considered to be extremely stable.

Figure 2.2 on the following page describes the complete process of applying wavefront scheduling to LLVM intermediate representation.

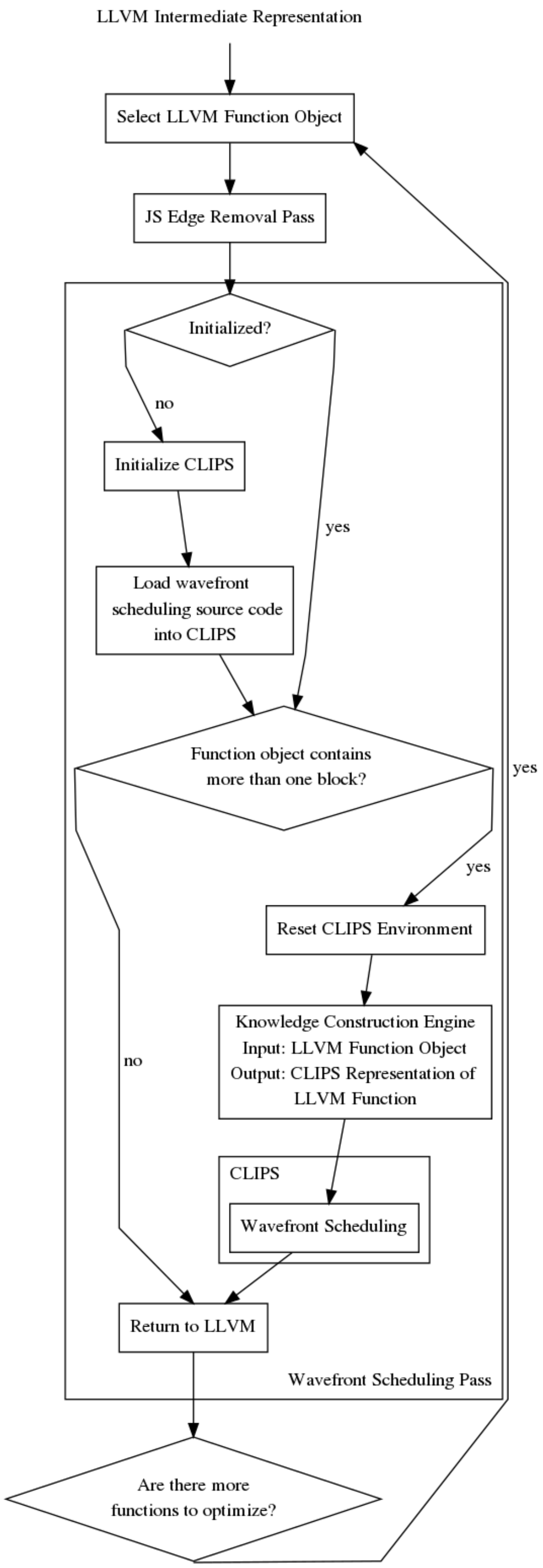
LLVM Intermediate Representation
Select LLVM Function Object
JS Edge Removal Pass
Initialized?
no
yes
Initialize CLIPS
Load wavefront scheduling source code into CLIPS
Function object contains more than one block?
yes
yes
no
Reset CLIPS Environment
Knowledge Construction Engine Input: LLVM Function Object Output: CLIPS Representation of LLVM Function
CLIPS
Wavefront Scheduling
Return to LLVM
Wavefront Scheduling Pass
Are there more functions to optimize?

Figure 2.2: The detailed view of how CLIPS and LLVM are integrated

Chapter 3

Wavefront Scheduling within CLIPS

This chapter will walk the reader through the entire act of wavefront scheduling starting when control is handed off to CLIPS by LLVM via the call to Run in the WavefrontScheduling pass. This chapter does not talk about *all* of the rules defined to implement wavefront scheduling because there are a number of them that are either meant for debugging purposes or are defined as a side effect of implementing the optimization within as an expert system. This means that only 270 of the nearly 310 rules will be discussed in this chapter[1].

The rules defined within CLIPS to carry out wavefront scheduling follow the paper [4] as much as possible but take liberties whenever necessary. This implementation uses the concept of a *control fact* to force the expert system to only execute rules that are part of a target *stage*. Normally, the description of how the implementation works would follow along the lines of these stages. However, following the stages would not only be confusing at times but also detrimental due to the fact that some of them were only introduced to ensure that the optimization operated correctly. To this end, it was decided that describing the operation of this optimization should follow the original paper as much as possible.

For the sake of convience, every single rule has been annotated with a unique number starting from one. The rules are broken up into several parts described by Table 3.1. These

[1]The source code will be published on the internet under the BSD license for all to see

actions will be discussed in sections 3.1 through 3.7 respectively.

Section	Action	Associated Rule Numbers
3.1	Control Fact Management	1,2
3.2	Fixup CFG	3-21
3.3	Dependency Analysis	22-88
3.4	Path Construction	89-105
3.5	Wavefront Initialization	106-112
3.6	Wavefront Scheduling	113-264
3.7	Advancing the Wavefront	265-270

Table 3.1: Breakdown of what rules are part of what section

The other aspect that needs to be addressed before ”diving” into the code are the classes used by the optimization. There are quite a number of classes defined in CLIPS to represent all of the objects built by the KCE as well as structures necessary to store different types of information necessary for wavefront scheduling. For the sake of brevity only the types that are considered necessary will be mentioned in Table 3.2 on the next page.

With this information it is now possible to talk about the process of wavefront scheduling starting with control fact management.

Class Name	Description
TaggedObject	An object that is the super class of all types defined for implementing wavefront scheduling. It exposes the ID and Parent fields which are used extensively throughout the implementation.
InteropObject	An object that contains the pointer address of the LLVM representation of the target object. This is one of the supertypes all classes defined for implementing wavefront scheduling.
Diplomat	An object that describes all dependency information for basic blocks, regions, and loops.
BasicBlock	The CLIPS representation of the LLVM BasicBlock class. In addition to being a User this class is also a Diplomat.
Region	The CLIPS representation of the LLVM Region class. This class is also a Diplomat.
Loop	The CLIPS representation of the LLVM Loop class. Unlike LLVM, the Loop class extends off of the Region class in CLIPS.
Instruction	Base type of all LLVM instructions. Extends off of the User class.
Path	An object meant to store a given path through a target region.
CompensationPathVector	A CLIPS implementation of the compensation path vector described in [4].
PathAggregate	An object that stores a large amount of different information for each block that is currently on the wavefront.
Wavefront	The CLIPS representation of the wavefront described in [4]. It keeps track of open and closed blocks on the wavefront.

Table 3.2: List of important classes

3.1 Control Fact Management

The first rule that is fired upon execution of CLIPS by LLVM has to do with the declaration of stages and moving between them. Since CLIPS is unordered in its execution, it is necessary to define a *control fact* as an *ordering* mechanism. The control fact is always the first fact of any rule and exploits the characteristic that any changes to the control fact cause anything dependent on it to be reactivated as well. The control fact is the first thing that is created upon control transferring from LLVM to CLIPS.

Rule 1: first-rule

```
(defrule first-rule
         (declare (salience 10000))
         (initial-fact)
         =>
         (assert (Stage Fixup FixupUpdate
                   FixupRename Analysis ExtendedMemoryAnalysis
                   Path PathUpdate WavefrontInit WavefrontSchedule
                   WavefrontFinal Final)))
```

This rule asserts the control fact that describes at what *stage* the optimization is currently. For the purpose of wavefront scheduling there are 11 stages: Fixup, FixupUpdate, FixupRename, Analysis, ExtendedMemoryAnalysis, Path, PathUpdate, WavefrontInit, WavefrontSchedule, WavefrontFinal, and Final. It is important to note that this rule will only fire once per call to Reset because the initial-fact is consumed by the firing of this rule. However, it is also necessary to be able to move to the next stage automatically. The second rule, change-stage, is fired after all other rules of a given stage have fired.

Rule 2: change-stage

```
(defrule change-stage
         (declare (salience -10000))
         ?fct <- (Stage ? $?rest)
         =>
         (retract ?fct)
         (assert (Stage $?rest)))
```

When a stage is finished completely, it is necessary to move to the next stage to continue the act of wavefront scheduling. This rule is a generic stage advancer that deletes the current stage and replaces it with the rest of the stage list. When the stage is advanced in this fashion, the agenda is reset which allows all objects and facts to become valid matching targets again. The salience is set to -10000 which ensures that it is the *last* rule fired in a given stage.

These two rules are not directly involved with the act of wavefront scheduling. Instead they are tasked with ensuring the smooth operation of wavefront scheduling. This would technically make them violations of the understanding that only rules critical to the act of wavefront scheduling would be described in this paper. However, in this case, the omission of these rules would add confusion because *every* single rule uses the Stage control fact.

It is now time to dive right into this implementation of wavefront scheduling. The first stop is fixing the CFG.

3.2 Correcting the CFG

One of the assumptions that [4] makes is that regions and loops are part of the same CFG. The paper does not actually explicitly state that loops and regions must be part of the same CFG but since *icc* does it automatically, it did not need to be stated. This requirement was discovered to be extremely critical during implementation with respect to determining paths through a given region, as LLVM does not provide a built-in mechanism to do this merging. Originally, another pass was going to be implemented to merge the two structures together before the KCE was invoked. However, it was found that this technique was not only very error prone but also difficult to maintain. Instead, it was decided to shift the task onto CLIPS due to the fact that its pattern matching capabilities would make such an action far easier to implement and test. Thus, LLVM's task became one of *informing* CLIPS of the merging through the assertion of facts during the process of converting regions into knowledge. These facts contain commands that transfer ownership of basic blocks from loops to regions meant to be nested within these loops. There are other tasks that this part is tasked with including:

1. The removal of redundant regions

2. Adding loops to target regions

3. Propagating a basic block's set of elements produced to its parents.

There are 18 rules that handle all of these tasks. The description of these rules will be broken up into two parts with the first being loop and region merging; and the second being the propagation of production information.

It is important to remember that the order these rules are described in is *not* the order in which they are fired, as the expert system is operates in an unordered fashion.

3.2.1 Merging Loops and Regions into a single CFG

The rules in this section follow what seems to be a linear order. However, this ordering is only to how the rules are executed. Remember, that the order with which facts and instances are matched is still unordered. The description of the act of merging loops and regions will start with the description of how the knowledge asserted by LLVM during knowledge construction is used. After that, the rule devoted to removing redundant regions is discussed. Finally, the rules associated with merging loops and regions into a single representation is explained.

Rule 3: ConvertControlModification

```
(defrule ConvertControlModification
         (Stage Fixup $?)
         ?fct <- (ControlModification (ModificationType Relinquish) (To ?r)
                                      (Subject $?elements))
         =>
         (retract ?fct)
         (assert (Put into ?r elements $?elements)))
```

This rule takes the template created by the KCE and converts it into a fact that is far easier to manage. This was done as a compatibility measure to prevent the need to rewrite and test a portion of the C++ side of the optimization. The fact asserted is used in what could be described as a *draining* fashion. That is where the first element of the list is used, the fact is retracted, and a new one is put in its place with the matched element missing. This is done to ensure that all elements described in the above template are operated on.

Rule 4: InjectElementsIntoNewBlock

```
(defrule InjectElementsIntoNewBlock
         (Stage Fixup $?)
         ?fct <- (Put into ?r elements ?element $?elements)
         ?region <- (object (is-a Region) (ID ?r))
         ?block <- (object (is-a BasicBlock) (ID ?element))
         =>
         (assert (Put into ?r elements $?elements))
         (modify-instance ?block (Parent ?r))
         (slot-insert$ ?region Contents 1 ?element)
         (retract ?fct))
```

This rule allows the target region to take control of the given basic block. The use of the slot-insert$ command is a way of quickly, and efficiently, adding an element to a multislot of a target object.

Rule 5: InjectElementsIntoNewBlock-Retract

```
(defrule InjectElementsIntoNewBlock-Retract
         (Stage Fixup $?)
         ?fct <- (Put into ?r elements)
         =>
         (retract ?fct))
```

Once all blocks destined for a target region have been moved, it is necessary to retract it to signify the process has been completed. There are a number of rules of this form through out the implementation to handle this kind of behavior.

Rule 6: ReplaceRegionWithLoop

```
(defrule ReplaceRegionWithLoop
         "Replaces a region with a loop if it turns out that the region is a
         container over a given loop (it is the only element in the region).
         The loop takes the place of the region in the grand scheme of things"
         (declare (salience -10))
         (Stage Fixup $?)
         ?r <- (object (is-a Region) (Class Region) (ID ?name) (Parent ?p)
                       (Contents ?loop))
         ?l <- (object (is-a Loop) (ID ?loop)) ;we don't care what the parent
             is
         ?parent <- (object (is-a Region) (ID ?p)
                            (Contents $?start ?name $?rest))
         =>
         (modify-instance ?l (Parent ?p))
         (modify-instance ?parent (Contents $?start ?loop $?rest))
         (unmake-instance ?r))
```

This rule is tasked with removing redundant regions from a given CFG. This means that the region only contains a loop. If this rule is not present, the optimization will still work, however it will consume more memory due to the extra regions that do not need to be there. This rule has the lowest priority of the Fixup stage because it is necessary that certain regions and loops are not deleted before a potential transfer of ownership can take place. The low priority nature of this rule also ensures that there are regions that only contain a loop because it is necessary to transfer blocks into the region first. When this is the case the region needs to be updated first so that the redundancy test does not remove it and cause an incorrect function representation to be generated.

The rest of the rules related to loop and region merging use what is known as subset conversion to correctly merge the disparagte elements. The idea behind subset conversion is that since the contents of each region and loop are already provided by LLVM it is only necessary to perform subset checks to find out if a region contains elements that actually belong to a subregion or loop.

Rule 7: ReplaceSubsets

```
(defrule ReplaceSubsets
         (Stage FixupUpdate $?)
         ?region <- (object (is-a Region) (ID ?r) (Contents $?a))
         ?loop <- (object (is-a Loop) (ID ?l&~?r) (Contents $?b))
         (test (subsetp ?a ?b))
         =>
         (assert (In ?l put ?r in place of $?a)))
```

This rule checks to see if a region is a subset of a given loop. If it turns out that this is the case then it will be necessary to replace the elements that make up the subset with the name of the region in question within the target loop. This need is not immediately acted upon and is instead deferred through the use of a fact. It is important to note that the sub region can also be a loop as well due to the fact that a loop is a sub class of region.

Rule 8: MergeElementsFromLoop

```
(defrule MergeElementsFromLoop
         (Stage FixupUpdate $?)
         ?fct <- (In ?l put ?r in place of ?first $?rest)
         ?loop <- (object (is-a Loop) (ID ?l) (Contents $?a ?first $?b))
         =>
         (retract ?fct)
         (assert (In ?l put ?r in place of $?rest))
         (modify-instance ?loop (Contents $?a $?b)))
```

This rule takes the fact describing the deferred action of replacing elements within the body of the target loop, with a given region and actually performs the action. This rule *drains* the fact of its elements one by one as the location of each element is not known, although it can be easily found through pattern matching. At this point, the region is not inserted into the loop's contents to prevent the generation of duplicates. This process continues until the fact is completely drained. There are two rules that handle the act of merging the target region into the given loop. Think of these rules to be the *or* to a conditional. Either one or the other will fire. Failure to have both rules could result in a fact not being matched and the function representation becomes incorrect because there is either a missing or extra region within the target loop.

Rule 9: MergeRegionInPlaceOfElementsToLoop

```
(defrule MergeRegionInPlaceOfElementsToLoop
         (Stage FixupUpdate $?)
         ?fct <- (In ?l put ?r in place of)
         ?loop <- (object (is-a Loop) (ID ?l))
         (test (eq FALSE (member$ ?r (send ?loop get-Contents))))
         =>
         (retract ?fct)
         (slot-insert$ ?loop Contents 1 ?r))
```

This rule is what could be considered to be the "default" action to be taken, as most of the time the region being inserted into the target loop is not already a part of said loop. This requires that not only does the fact be retracted but the name of the region be added to the loop's contents.

Rule 10: MergeRegionInPlaceOfElementstoLoop-NoDuplicate

However, sometimes it turns out that the loop already contains the region in question. When this happens only the fact is removed. The condition for this rule differs only in that the test checks to see if the given region is already a member of the target loop.

Sometimes, it turns out that at this point there are regions and loops that contain basic blocks, regions or loops in the Contents multislot that are not actually a parent of the said element. Solving this problem requires further use of subset conversion.

Rule 11: StripoutUncontrolledElements

```
(defrule StripoutUncontrolledElements
         (declare (salience -2))
         (Stage FixupUpdate $?)
         (object (is-a Region) (ID ?r) (Contents $? ?e $?))
         (object (is-a TaggedObject) (ID ?e) (Parent ?z&~?r))
         =>
         (assert (Parent of ?e is now ?r)
                 (Rename: In ?z put ?r in place of ?e)))
```

When a region contains an element[2] who's parent is not the region itself, it is necessary to not only remove the element in question, but also replace it with the element's actual parent. This rule has lower priority because it is necessary to make sure that all loop and region merging has taken place before analysis is performed to figure out what elements need to be removed and replaced to bring the representation of the target function back into the realm of correctness.

The next set of rules is meant to take the facts asserted by rule 11 and use them to fix the CFG. Since this is an expert system there are states that have to explicitly handled that would become the *else* clause in other programming languages. In this case the *else* clause becomes the first rule discussed.

Rule 12: RetractImpossibleReplaceStatements

```
(defrule RetractImpossibleReplaceStatements
         (declare (salience 1))
         (Stage FixupRename $?)
         ?f0 <- (Rename: In ?r put ?t in place of $?)
         (not (exists (object (ID ?r))))
         =>
         (retract ?f0))
```

This rule will delete facts denoting a required rename operation as impossible if it turns out that the target region actually refer to an instance of an object. This rule only fires when dealing with top-level regions and loops because the function itself is not represented by an object within CLIPS. However, the parent of these regions and loops is still set to the name of the function itself. Since the value stored within the Parent field is a symbol

[2]This can be a basic block, region, or loop.

instead of a instance-name or address, it is necessary to check to make sure that the symbol refers to an actual object. This rule is also given higher prioirty to ensure that impossible replace statements are removed immediately after this stage starts.

However, even if the target region does have an object associated with it, it is necessary to reduce the number of messages sent to a given object by merging rename facts into one long version.

Rule 13: MergePutStatements

```
(defrule MergePutStatements
         (declare (salience 1))
         (Stage FixupRename $?)
         ?f0 <- (Rename: In ?r put ?t in place of $?e0)
         ?f1 <- (Rename: In ?r put ?t in place of $?e1)
         (test (neq ?f0 ?f1))
         =>
         (retract ?f0 ?f1)
         (assert (Rename: In ?r put ?t in place of $?e0 $?e1)))
```

This rule is quite simple as it takes two different rename facts that refer to the same object and merge the corresponding list of elements into a single fact. The test to make sure the facts are not the same prevents a fact from being matched twice in the same rule. Once all mergers have been completed the number of facts will obviously be reduced to a far more manageable size that can be used in a draining capacity as denoted below.

Rule 14: RevokeOwnershipOfIllegalElements

```
(defrule RevokeOwnershipOfIllegalElements
         (Stage FixupRename $?)
         ?fct <- (Rename: In ?r put ?t in place of ?first $?rest)
         ?region <- (object (is-a Region) (ID ?r)
           (Contents $?before ?first $?after))
         =>
         (retract ?fct)
         (assert (Rename: In ?r put ?t in place of $?rest))
         (modify-instance ?region (Contents $?before $?after)))
```

This rule takes an element from a rename fact and deletes it from the contents of the target region. Obviously, this rule takes a draining approach to this. This is necessary because the index of the target element is not known.

Rule 15: ReplaceOwnershipOfIllegalElements-Final-InsertAndRetract

```
(defrule ReplaceOwnershipOfIllegalElements-Final-InsertAndRetract
         (Stage FixupRename $?)
         ?fct <- (Rename: In ?r put ?t in place of)
         ?region <- (object (is-a Region) (ID ?r))
         (test (eq FALSE (member$ ?t (send ?region get-Contents))))
         =>
         (retract ?fct)
         (slot-insert$ ?region Contents 1 ?t))
```

Once a rename fact has been drained it is necessary to insert the replacement target into the corresponding region. This rule handles the case where the replacement target is not a member of the contents multislot of the corresponding region.

Rule 16: ReplaceOwnershipOfIllegalElements-Final-JustRetract

This is the dual of the previous rule that only retracts the rename fact because the element ?t is already a member of the Contents multislot of region ?r.

The next rule is the last rule dealing with the act of merging loops and regions together into one CFG.

Rule 17: ReplaceParentOfGivenItem

```
(defrule ReplaceParentOfGivenItem
         (Stage FixupRename $?)
         ?fct <- (Parent of ?t is now ?r)
         ?o0 <- (object (ID ?t))
         =>
         (retract ?fct)
         (modify-instance ?o0 (Parent ?r)))
```

This rule is responsible with handling the other fact asserted in rule 11. It is quite straightforward in what it is does. However, the reason behind it may be a little mystifying. Sometimes, it turns out that the an element may be part of the proper parent region or loop, but the Parent field of the target element does not reflect this. Thus the fact asserted in rule 11 ensures that this information is properly reflected.

3.2.2 Propagation of Producers

The other task that this part of wavefront scheduling is responsible for involves the propagation of the Produces multislot of each BasicBlock to its parents. It is also responsible for identifying the set of non local dependencies which is used extensively during the actual act of wavefront scheduling.

Rule 18: PropagateBlockProducers

```
(defrule PropagateBlockProducers
         (declare (salience 100))
         (Stage Fixup $?)
         (object (is-a BasicBlock) (ID ?b) (Parent ?r)
                 (Produces $?produces))
         =>
         (assert (Give ?r from ?b the following produced items $?produces)))
```

This rule takes the set of elements Produced from the given basic block, and propagates this information to the parent of the given block. It is not necessary to check if the parent of a basic block exists because a basic block will *always* have a parent. This rule starts the process of propagation off, however it is up to the next two rules to actually continue the process.

Rule 19: PropagateRegionProducers-ParentExists

```
(defrule PropagateRegionProducers-ParentExists
         (declare (salience 50))
         (Stage Fixup $?)
         ?fct <- (Give ?r from ? the following produced items $?produced)
         ?region <- (object (is-a Region) (ID ?r) (Parent ?p))
         (exists (object (is-a Region) (ID ?p)))
         =>
         (retract ?fct)
         (assert (Give ?p from ?r the following produced items $?produced))
         (slot-insert$ ?region Produces 1 ?produced))
```

This rule takes the propagation information and adds it to the Produces list of the target region. It then asserts a fact of the same form that targets the parent of the current region, which starts the process all over again. This rule fires if the parent of the target region actually exists.

Rule 20: PropagateRegionProducers-ParentDoesntExist

However, when the parent of the target region does not actually exist then this rule catches the fact and does not allow the propagation to go any further. The main differences between this rule and the previous one, are that the existence test has a not around it, and there is no fact assertion. Eventually, all facts will reach this rule and the process will terminate. Once all facts have been retracted and all of this information has been created, it is necessary to determine the set of non local dependencies for each instruction currently defined.

Rule 21: IdentifyNonLocalDependencies

```
(defrule IdentifyNonLocalDependencies
         (Stage Fixup $?)
         ?i0 <- (object (is-a Instruction) (Parent ?p) (ID ?t0)
                        (Operands $? ?op $?))
         (object (is-a Instruction) (ID ?op) (Parent ~?p))
         (test (eq FALSE (member$ ?op (send ?i0 get-NonLocalDependencies))))
         =>
         (slot-insert$ ?i0 NonLocalDependencies 1 ?op))
```

This rule is actually independent from the previous three rules so any attempts to see a connection are just a waste of time. The only part of this rule that is important to note is that it employs a check to only add operands to the NonLocalDependencies multislot of the target instruction if it is not a member. This ensures that the multislot remains a set which is absolutely critical during the act of wavefront scheduling.

One may be curious as to why this rule is even in this stage as it has nothing to do with the overall objective of this part. Well this rule is here to ensure that non-local dependencies are generated in an environment that does not operate on instructions. If this rule were added to the dependency analysis part of wavefront then it would be necessary to introduce a stage that would contain only this rule. The number of times this rule fires is correlated directly to the number of instructions within the function being currently analyzed.

3.3 Analyzing the target function

Before performing wavefront scheduling and generating paths it is necessary to

1. Identify which regions are viable candidates for wavefront scheduling

2. Identify dependencies between instructions

3. Attempt to identify so-called memory barriers to make the identification of instructions that can be scheduled easier.

4. Attempt to identify where load and store instructions read from and write to respectively.

These actions are absolutely critical to the act of wavefront scheduling. However, the paper [4] itself does not talk about any of this. The most likely explaination is that it is up to the implementer to realize that these actions will be necessary. It could also be that the paper [4] describes how it is done within *icc*. This means that implementation details would be left out due to the fact that *icc* is closed-source software.

3.3.1 Determining Viable Wavefront Scheduling Candidates

The first act that the analysis section performs is determining whether a given region is a viable candidate for wavefront scheduling. Sometimes, LLVM will select regions that consist solely of a single block. These single block regions are actually a huge waste of computing resources because the amount of extra work to schedule that single block was deemed to be overkill. Thus it was necessary to go through each region and find out how many blocks given region has. If the number of blocks is equal to one then the given region is not a viable candidate for wavefront scheduling and should be skipped.

Rule 22: InstanceFrequencyCounter

```
(defrule InstanceFrequencyCounter
         "Creates a frequency counter hint for basic blocks"
         (declare (salience 220))
         (Stage Analysis $?)
         (object (is-a Region) (Class Region) (ID ?p)
                 (CanWavefrontSchedule FALSE))
         =>
         (make-instance (gensym*) of FrequencyAnalysis (Parent ?p)
                        (Point BasicBlockCount)))
```

This rule selects regions that are not loops which is denoted through the use of the Class slot and constructs a FrequencyAnalysis object which will keep track of the number of blocks for the target region. Loops are ignored because it was not known at the time of implementation how one would construct the set of paths through the loop. Thus loops are considered to not be valid wavefront scheduling targets because of this.

Rule 23: IncrementFrequencyCounter-BasicBlock

```
(defrule IncrementFrequencyCounter-BasicBlock
         "Goes through a given Region counting the number of basic blocks
             found
         within the region. Valid blocks are blocks that contain more than one
         instruction as we don't want to count JS nodes as they don't usually
         contain code."
         (declare (salience 210))
         (Stage Analysis $?)
         (object (is-a Region) (ID ?p)
                 (CanWavefrontSchedule FALSE)
                 (Contents $? ?t $?))
         ?bb <- (object (is-a BasicBlock) (ID ?t) (Parent ?p))
         ?fa <- (object (is-a FrequencyAnalysis) (Parent ?p)
                        (Point BasicBlockCount))
         (test (> (length$ (send ?bb get-Contents)) 1))
         =>
         (send ?fa .IncrementFrequency))
```

This rule is the so-called *meat* of the process to determine if the region is a viable candidate. Normally, a region with a single block would be accurately identified. However, with the addition of the pass to remove JS edges, those regions that once consisted of a single block now will most likely contain multiple blocks due to the inclusion of JS nodes. The problem with JS nodes is that they do not contain any instructions that can be scheduled. Thus the counting process was changed to only count basic blocks that contain more than one instruction. It is important to understand that a blank JS node still contains the terminator instruction. While it has been mentioned several times that message passing slow, it is sometimes necessary as is the case in this rule.

Rule 24: ImplyEnoughBlocks

```
(defrule ImplyEnoughBlocks
        "There are enough blocks within the target region to make it a
        candidate for wavefront scheduling. Make a hint that says this."
        (declare (salience 200))
        (Stage Analysis $?)
        ?fa <- (object (is-a FrequencyAnalysis) (Parent ?p)
                       (Point BasicBlockCount)
                       (Frequency ?z&:(and (< ?z 100) (> ?z 1))))
        ?region <- (object (is-a Region) (ID ?p))
        =>
        (unmake-instance ?fa)
        (modify-instance ?region (CanWavefrontSchedule TRUE)))
```

Once all elements of a given region have been analyzed, it is time to actually determine if the region is a viable wavefront scheduling candidate. As stated earlier, if the region only has one block then it is not a viable candidate for wavefront scheduling. However, it was determined that having a region with more than 100 blocks was also detrimental to the wavefront scheduling process as well. If the number of blocks exceeded 100 then the scheduling process for open blocks on the wavefront would get slower and slower for each advancement. This is the reason for the inclusion of the upper bound. This slow down was perplexing and upon rereading the paper [4] it was found that there was a mention of the fact that *icc* operates on specially formed scheduling regions that usually consist of around seven blocks[4]. This discovery not only justifies the bounds but also brings the implementation closer in line with the paper [4].

These rules use the salience command in place of a separate stage to force them to fire before the rest of the rules in the analysis section. This was done for the sake of brevity more than anything else.

Once the set of regions that are deemed viable for wavefront scheduling have been selected, it is time to generate the list of producers and consumers for all instructions in the given function.

3.3.2 Computing Instruction Dependencies

Computing dependencies is an important part of wavefront scheduling because it ensures that instructions are scheduled in the proper order into blocks on the wavefront. These dependencies are defined for each instruction through a list of producers and consumers. The producers list contains all instructions that *produce* the given instruction. Whereas, the consumers list contains all instructions that consume the value generated by the given instruction. There are several kinds of instruction dependencies: WAR, RAW, WAW, Call, and Pre-Call. While the first three have already been described the last two have not.

A call dependency[3] is a special kind of dependency that is meant to ensure correctness when: (1) a given call instruction modifies memory, (2) is inline assembler, (3) or has side effects. Every instruction, in the basic block, that follows the call instruction has the call instruction marked as a producer. This is an artificial dependency created to sidestep the need to perform memory analysis on each call which is both time consuming and error-prone.

The other dependency type, Pre-Call, is a dependency designed to ensure that call instructions are dependent not only on previous call instructions but any instructions that come before it. This ensures that the serial order of the instructions is maintained even if instructions are scheduled earlier in the function.

These rules were constructed to answer the question. "Can I execute these two instructions in parallel safely?". These rules are meant to *explain* the cases where the answer is *no* as the number of failure cases is quite small compared to the number of cases answering *yes*. This means that groups of instructions that do not match will be considered to not have any dependencies between one another. It should also be noted that these dependencies are only identified between instructions in the same basic block. This is done because computing all dependencies between all instructions is quite expensive, and has already been done in the previous section for the computation of non-local dependencies.

The dependency information is cataloged in each instruction is described through a list of producers and consumers. The list of producers represent the values that have be computed before this instruction can *produce* its result. The list of consumers represent the list of instructions that *consume* the value produced by the instruction. These lists are quite flexible and are used twice during the act of wavefront scheduling. The first time is when instructions are being scheduled into blocks on the wavefront. The second time is

[3]Also known as a post-call dependency

when a block on the wavefront has basic block scheduling applied to it.

Rule 25: IdentifyWAR

```
(defrule IdentifyWAR
        "Identifies a WAR dependency between two instructions. It will not
        match if it turns out the values are constant integers or constant
        floating point values"
        (Stage Analysis $?)
        ?i0 <- (object (is-a Instruction) (Parent ?p) (ID ?t0)
                       (SourceRegisters $? ?c $?) (TimeIndex ?ti0))
        (object (is-a TaggedObject&~ConstantInteger&~ConstantFloatingPoint)
                (ID ?c))
        ?i1 <- (object (is-a Instruction) (Parent ?p) (ID ?t1)
                       (TimeIndex ?ti1&:(< ?ti0 ?ti1))
                       (DestinationRegisters $? ?c $?))
        =>
        (assert (Instruction ?t1 consumes ?t0)
                (Instruction ?t0 produces ?t1)))
```

This rule handles the detection of a WAR hazard. In addition to comparing the source registers of the first instruction to the destination registers of the second it also makes sure that the first instruction has a lower time index than the second. This ensures that the serial ordering of the given instructions is maintained, so that the code will maintain correctness in the fact of being rescheduled. The test to make sure that the target operand is not a constant integer, or constant floating point, prevents the creation of an artificial dependency.

Rule 26: IdentifyRAW

This rule is extremely similar to IdentifyWAR. The only difference is that the destination registers of the first instruction are compared to the source registers of the second. Obviously, if the value is found in the source registers of the second then the second instruction needs to have the first instruction completely executed before it is safe to execute the second.

Rule 27: IdentifyWAW

This rule is extremely similar to IdentifyWAR and IdentifyRAW. The only difference is that the destination registers of the first instruction are compared to the destination registers of the second. This rule will only fire on store instructions as they represent the only way that LLVM writes to memory. These three rules were taken from an implementation of basic block scheduling that uses CLIPS. It was only slightly modified to include the check to prevent dependencies between instructions being created due to the instructions using the same constant values.

The following two instructions represent the identification of pre-call dependencies. These rules have the share the same basic structure and because of this, only the code associated with the first rule is shown. Each rule handles a different type of pre-call dependency that must be flagged.

Rule 28: MarkInstructionsThatHappenBeforeCall-WritesToMemory

```
(defrule MarkInstructionsThatHappenBeforeCall-WritesToMemory
         (Stage Analysis $?)
         (object (is-a CallInstruction) (ID ?n0) (Parent ?p)
                 (MayWriteToMemory TRUE) (TimeIndex ?t0))
         (object (is-a Instruction) (ID ?n1) (Parent ?p)
                 (TimeIndex ?t1&:(> ?t0 ?t1)))
         =>
         (assert (Instruction ?n0 consumes ?n1)
                 (Instruction ?n1 produces ?n0)))
```

This rule handles the case where the instruction that comes before the target call instruction must *produce* the call instruction because the call instruction writes to memory. Since it is important to preserve the order of memory modification, this rule ensures that the instructions that happen before the call are executed before the call instruction itself.

Rule 29: MarkInstructionsThatHappenBeforeCall-HasSideEffects

If it turns out that the call instruction has side effects, then it is necessary to ensure that any instructions that come before the call are executed ahead of it. This is because the side effects from calling the function may modify memory in such a way that causes invalidations to occur.

For those that are wondering, the fields used to create these dependencies (MayWrite-ToMemory, MayHaveSideEffects, etc) were populated by the KCE before control was passed over to CLIPS.

The next three rules describe the creation of post-call dependencies that ensure that any instructions that follow a call instruction that matches one of these rules is executed only after the instruction is completed. These three rules assert a fact that denotes the given block has a call barrier. What this call barrier does is prevent any instruction that is below the call from being moved above it to ensure that the code remains correct. This applies not only to the parent block in question but also any blocks that follow it as well.

Instead of matching each instruction individually, the following rules make use of pattern matching to grab the names of all instructions that follow the call instruction in question. This is a tad faster because it is not necessary to actually match against the time index of the target call instruction.

Rule 30: MarkCallInstructionDependency-ModifiesMemory

```
(defrule MarkCallInstructionDependency-ModifiesMemory
         "Creates a series of dependencies for all instructions following a
         call instruction if it turns out that the call could modify memory."
         (Stage Analysis $?)
         (object (is-a CallInstruction)
                 (ID ?name)
                 (Parent ?p)
                 (DoesNotAccessMemory FALSE)
                 (OnlyReadsMemory FALSE)
                 (MayWriteToMemory TRUE)
                 (TimeIndex ?t0))
         ?bb <- (object (is-a BasicBlock) (ID ?p) (Parent ?r)
                        (Contents $? ?name $? ?following $?))
         =>
         (assert (Region ?r has a CallBarrier)
                 (Block ?p has a CallBarrier)
                 (Instruction ?following has a CallDependency)
                 (Instruction ?following consumes ?name)
                 (Instruction ?name produces ?following)))
```

Identifying post-call dependencies are necessary to create because the instructions that follow the call may be dependent on the fact that it modifies a specific section of memory. While it may be believed that writing functions that operate in this fashion is a bad practice, it is not up to the compiler or optimizations to attempt to fix this. Instead, it needs to ensure that the code works correctly as it was described to the compiler.

Rule 31: MarkCallInstructionDependency-InlineAsm

If it turns out that the call instruction consists of inline assembler, then a call barrier must be erected to ensure that the assembly being executed will be correct. It is way beyond the scope of this optimization to analyze machine specific assembly to figure out what it is doing. Even if changes are made to this implementation in the future that safely allow the scheduling of call instructions, the call barrier erected by the use of inline assembly will remain.

Rule 32: MarkCallInstructionDependency-SideEffects

This rule handles the generation of post call dependencies if the call instruction has side effects. This is necessary to automatically preserve correctness if it turns out that there are instructions following the target call that are dependent on those side effects. However, it is also possible that the side effects do not actually write to memory, in this case the call dependency should not exist because there is nothing being modified. However, in general practice when a call has side effects it may write to memory as well.

Once, all dependencies have been found for all instructions in each basic block of the target function, it is necessary to merge these separate facts into one ”long” fact. There are six rules that govern this action. Only three of the six rules will have their code displayed because the difference between them is that one operates on consumers and the other operates on producers.

Rule 33: MergeConsumers

```
(defrule MergeConsumers
        (declare (salience -2))
        (Stage Analysis $?)
        ?f0 <- (Instruction ?a consumes ?id)
        ?f1 <- (Instruction ?b consumes ?id)
        (test (neq ?f0 ?f1))
        =>
        (retract ?f0 ?f1)
        (assert (Instruction ?id is consumed by ?a ?b)))
```

This rule is quite similar to the merging rule found in the section devoted to fixing the CFG. The only difference is that the layout of these rules necessitates the creation of a new fact so that the matched parameters can be reversed.

Rule 34: MergeProducers

As stated earlier this is the producer version of MergeConsumers. These two rules will continue to fire as long as a *singular* version of the produces or consumes fact remains. As the build up process continues, the next two rules step in to merge the multifield version of the produces and consumes facts to eventually reach a single fact describing all producers and consumers.

Rule 35: MergeConsumers-Multi

```
(defrule MergeConsumers-Multi
         (declare (salience -2))
         (Stage Analysis $?)
         ?f0 <- (Instruction ?id is consumed by $?a)
         ?f1 <- (Instruction ?id is consumed by $?b)
         (test (neq ?f0 ?f1))
         =>
         (retract ?f0 ?f1)
         (assert (Instruction ?id is consumed by $?a $?b)))
```

This rule is quite similar to the singular version except that it operates on a multifield instead of a single element. This merger can be quite expensive if there are enough elements because a copy of each multifield must be made during the assertion of the new fact.

Rule 36: MergeProducers-Multi

This is the producers version of the previous rule.

There is one extra case that must be handled because the rules that follow this merging will only operate on the multifield variant of the produces and consumes fact.

Rule 37: MergeConsumers-Only

```
(defrule MergeConsumers-Only
         (declare (salience -3))
         (Stage Analysis $?)
         ?f0 <- (Instruction ?a consumes ?b)
         =>
         (retract ?f0)
         (assert (Instruction ?b is consumed by ?a)))
```

Sometimes it turns out that a given value is consumed by only one instruction. This will bypass the previous four rules because of its singular nature. This rule will just rearrange the fact to simulate the multifield version even though there is only one element.

Rule 38: MergeProducers-Only

Obviously, this is the producers version of MergeConsumers-Only. These rules are necessary to wavefront scheduling because without them the list of producers and consumers is not correct. Which can lead to the occurence of either infinite loops or incorrect scheduling that the LLVM module verifier may not catch.

Once these rules have been completely fired it is now possible to actually add this new list of consumers and producers to the corresponding instruction.

Rule 39: InjectConsumers

```
(defrule InjectConsumers
         "Adds a given consumer to the target instruction"
         (declare (salience -6))
         (Stage Analysis $?)
         ?fct <- (Instruction ?id is consumed by $?insts)
         ?inst <- (object (is-a Instruction) (ID ?id))
         =>
         (retract ?fct)
         (slot-insert$ ?inst Consumers 1 ?insts))
```

Simply put, this rule adds the list of consumers to the instruction's Consumers multislot.

Rule 40: InjectProducers

This rule does the almost exact same thing as InjectConsumers except it is for the Producers multislot, and it adds the same list of producers to the LocalDependencies multislot of the target instruction. The set of local dependencies is used by wavefront scheduling to determine the order in which instructions should be scheduled into a block on the wavefront.

The next four rules handle updating regions, basic blocks, and instructions with the knowledge that they have a call dependency or call barrier.

Rule 41: FlagCallBarrierForRegion-ImbueParent

```
(defrule FlagCallBarrierForRegion-ImbueParent
         "Marks the given region has having a call barrier"
         (declare (salience -10))
         (Stage Analysis $?)
         ?fct <- (Region ?r has a CallBarrier)
         ?region <- (object (is-a Region) (ID ?r) (Parent ?p))
         (exists (object (is-a Region) (ID ?p)))
         =>
         (retract ?fct)
         (assert (Region ?p has a CallBarrier))
         (modify-instance ?region (HasCallBarrier TRUE)))
```

This rule marks the target region as having a call barrier and also asserts a fact to propagate this information to its parent as well. This rule will capture those newly asserted facts as well. Obviously, this rule will fire only if the region has object representation of the symbolic name of the parent.

Rule 42: FlagCallBarrierForRegion

This rule handles the case where the target region does not have an object bound to the symbolic name of its parent. It differs in that it does not assert a new fact as that does not make sense.

Rule 43: FlagCallBarrierForBasicBlock

```
(defrule FlagCallBarrierForBasicBlock
         "Marks the given region has having a call barrier"
         (declare (salience -10))
         (Stage Analysis $?)
         ?fct <- (Block ?r has a CallBarrier)
         ?block <- (object (is-a BasicBlock) (ID ?r) (Parent ?p))
         =>
         (retract ?fct)
         (assert (Region ?p has a CallBarrier))
         (modify-instance ?block (HasCallBarrier TRUE)))
```

If a basic block turns out to have a call barrier then it is necessary to imbue that basic block with that knowledge. This rule will start the propagation this call barrier to its parent region as well.

Rule 44: MarkHasACallDependency

```
(defrule MarkHasACallDependency
         (Stage Analysis $?)
         ?fct <- (Instruction ?target has a CallDependency)
         ?inst <- (object (is-a Instruction) (ID ?target))
         =>
         (retract ?fct)
         (if (not (send ?inst get-HasCallDependency)) then
           (modify-instance ?inst (HasCallDependency TRUE))))
```

Instructions are different from basic blocks and regions in that they do not have call barriers. Instead they are marked as having a call *dependency*. In the grand scheme of things it really is not important which instruction it is dependent on, because having such a dependency will prevent it from being scheduled into blocks on the wavefront. This rule does a check in the body to only set the instruction as having a call dependency if it is not marked as having one.

It is necessary to ensure that the set of producers, consumers, and local dependencies for each instruction was a set, so that any updates made during wavefront scheduling would be simple and quick to make. The next three rules convert the associated multislots to a set if is not one already.

Rule 45: SetifyInstructionProducers

```
(defrule SetifyInstructionProducers
         (declare (salience -11))
         (Stage Analysis $?)
         ?inst <- (object (is-a Instruction) (Producers $?a ?b $?c ?b $?d))
         =>
         (modify-instance ?inst (Producers $?a ?b $?c $?d)))
```

This rule converts the list of producers for a given instruction into a set. This rule is somewhat slow due to the pattern matching and continual updating of the target instruction.

Rule 46: SetifyInstructionConsumers

This rule handles the conversion of the list of consumers for a given instruction into a set. It is exactly the same as the previous instruction. The only difference is that the Producers multifield has been replaced with Consumers.

Rule 47: SetifyLocalDependencies

This rule is different from the previous two in that it targets the LocalDependencies multislot of the target instruction. It is probably the most important multifield to convert into a set.

These rules have described the actions that have to be taken for generating the set of dependencies between instructions. The only actions left to perform in this section of the implementation is to determine where memory barriers exist in the function being scheduled.

3.3.3 Identifying Memory Barriers

The identification of memory barriers is a quick way for wavefront scheduling to determine if a given instruction can be scheduled into a block on the wavefront. There are a considerable number of rules defined to identify memory barriers. However, it is important to note that the rules do not perform *infinitely* deep pointer analysis due to the fact that it was deemed to be a project within itself to implement. This led to the introduction of the UNKNOWN symbol that prevents store and load instructions from being scheduled above basic blocks, regions, loops, and instructions that write to it.

There are two rules that start off memory barrier analysis. Each rule is devoted to analysis of load or store instructions with each of them working a little differently than the other.

Rule 48: AssertLoadBarrierEvaluation

```
(defrule AssertLoadBarrierEvaluation
         "Creates a fact that tells the system to analyze the given load
         instruction to see if it is necessary to create a memory barrier for
         the given instruction"
         (declare (salience 5))
         (Stage Analysis $?)
         (object (is-a LoadInstruction) (ID ?t0) (SourceRegisters ?target $?))
         =>
         (assert (Analyze ?target for load ?t0)))
```

This rule takes the SourceRegisters of the target load instruction and analyzes only the first element. This is done because the address the load instruction will read from is stored in a register or will be computed by a constant address in memory described through the use of a ConstantExpression. Since there is only one instance of a constant expression it is possible to use it as though its a register. The fact asserted by this rule requests that the target operand is analyzed for the given load instruction.

Rule 49: AssertStoreBarrierEvaluation

```
(defrule AssertStoreBarrierEvaluation
         "Creates a fact that tells the system to analyze the given store
         instruction to see if it is necessary to create a memory barrier for
         it"
         (declare (salience 5))
         (Stage Analysis $?)
         (object (is-a StoreInstruction) (ID ?t0)
                 (DestinationRegisters ?target))
         =>
         (assert (Analyze ?target for store ?t0)))
```

When a store instruction is, found the destination register is selected as the address that the given value is going to be written to in memory.

The facts asserted by these two rules are used by the next 11 rules to determine memory barriers. Most of these rules attempt to identify the exact location being written to or read from in memory. However, this analysis will only resolve a single layer of indirection. Anything beyond that will cause the value UNKNOWN to be set as the target which generates a memory barrier.

Rule 50: PopulateBasicBlockWithReadFrom-Alloca

```
(defrule PopulateBasicBlockWithReadFrom-Alloca
         "Does a check to see if the load instruction refers directly to an
         AllocaInstruction or Constant. If it does then mark it as though the
         block reads from it"
         (declare (salience 4))
         (Stage Analysis $?)
         ?fct <- (Analyze ?target for load ?t0)
         (object (is-a LoadInstruction) (ID ?t0) (Parent ?p))
         (object (is-a AllocaInstruction) (ID ?target))
         =>
         (retract ?fct)
         (assert (instruction ?t0 memory target ?target)
                 (block ?p reads from ?target)))
```

If the given load instruction directly references an alloca then it means that the exact location of block of memory is being read from is known.

Rule 51: PopulateBasicBlockWithWriteTo-Alloca

When dealing with store instructions a direct reference to an alloca instruction means that the memory location that is being written to is known.

Rule 52: PopulateBasicBlockWithReadFrom-Constant

```
(defrule PopulateBasicBlockWithReadFrom-Constant
         "Does a check to see if the load instruction refers directly to an
         AllocaInstruction or Constant. If it does then mark it as though the
         block reads from it"
         (declare (salience 4))
         (Stage Analysis $?)
         ?fct <- (Analyze ?target for load ?t0)
         (object (is-a LoadInstruction) (ID ?t0) (Parent ?p))
         (object (is-a Constant) (ID ?target))
         =>
         (retract ?fct)
         (assert (instruction ?t0 memory target ?target)
                 (block ?p reads from ?target)))
```

This rule prevents the generation of a memory barrier if it turns out that the memory location being read from has been defined by a constant. Obviously, this means that it will not change and as such the memory target of this load instruction will be known

Rule 53: PopulateBasicBlockWithWriteTo-Constant

The same logic applies to store instructions that write to a constant address. These four rules are handle the cases where the location being written to is directly known.

The next seven rules handle the single layer of indirection that this implementation currently handles. The address of a pointer is indirectly defined through the use of the getlementptr instruction. It is important to note that getelementptr does not modify memory and is tasked with only computing the address to load from or store to.

Rule 54: IdentifyGetElementPointerLoadBarrier

```
(defrule IdentifyGetElementPointerLoadBarrier
        "Creates a load memory barrier hint if it turns out that the load
        instruction refers to a getelementptr instruction but that
        getelementptr doesn't refer to an alloca instruction or constant."
        (declare (salience 4))
        (Stage Analysis $?)
        ?fct <- (Analyze ?target for load ?t0)
        (object (is-a LoadInstruction) (ID ?t0) (Parent ?p))
        (object (is-a GetElementPointerInstruction) (ID ?target)
                (SourceRegisters ?a $?))
        (object (is-a ~AllocaInstruction&~Constant) (ID ?a))
        (object (is-a BasicBlock) (ID ?p) (Parent ?r))
        =>
        (retract ?fct)
        (assert (block ?p reads from UNKNOWN)
                (instruction ?t0 memory target UNKNOWN)
                (Block ?p has a MemoryBarrier)
                (Region ?r has a MemoryBarrier)))
```

This rule is fired if it turns out that the value referenced by the getelementptr instruction tied to a given load is neither an alloca instruction or constant instruction. This means that either the pointer address was computed dynamically, or there are more layers of indirection that have to be resolved. In either case, this rule handles the generation of a memory barrier to compensate for this lack of information.

Rule 55: IdentifyGetElementPointerStoreBarrier

The same actions are taken place when the target of a given store instruction is a getelementptr instruction that does not directly refer to a alloca instruction or constant. The only difference between the two instructions is that the facts asserted in this rule are tailored towards writes instead of reads.

Rule 56: PopulateBasicBlockWithReadFrom-GetElementPointer-Alloca

```
(defrule PopulateBasicBlockWithReadFrom−GetElementPointer−Alloca
         "Does a check to see if a given LoadInstruction referring to a
         GetElementPointerInstruction refers to an AllocaInstruction or
         Constant. If it does then mark it in the ReadsFrom multifield"
         (declare (salience 4))
         (Stage Analysis $?)
         ?fct <− (Analyze ?target for load ?t0)
         (object (is−a LoadInstruction) (ID ?t0) (Parent ?p))
         (object (is−a GetElementPointerInstruction) (ID ?target)
                 (SourceRegisters ?a $?))
         (object (is−a AllocaInstruction) (ID ?a))
         =>
         (retract ?fct)
         (assert (instruction ?t0 memory target ?a)
                 (block ?p reads from ?a)))
```

Rule 57: PopulateBasicBlockWithWriteTo-GetElementPointer-Alloca

However, if the address referred to by the getelementptr is an alloca instruction then the corresponding memory target is directly known. These rules do the same thing but for loads and stores respectively.

Rule 58: PopulateBasicBlockWithReadFrom-GetElementPointer-Constant

```
(defrule PopulateBasicBlockWithReadFrom−GetElementPointer−Constant
         "Does a check to see if a given LoadInstruction referring to a
         GetElementPointerInstruction refers to an AllocaInstruction or
         Constant. If it does then mark it in the ReadsFrom multifield"
         (declare (salience 4))
         (Stage Analysis $?)
         ?fct <− (Analyze ?target for load ?t0)
         (object (is−a LoadInstruction) (ID ?t0) (Parent ?p))
         (object (is−a GetElementPointerInstruction) (ID ?target)
                 (SourceRegisters ?a $?))
         (object (is−a Constant) (ID ?a))
         =>
         (retract ?fct)
         (assert (instruction ?t0 memory target ?a)
                 (block ?p reads from ?a)))
```

Rule 59: PopulateBasicBlockWithWriteTo-GetElementPointer-Constant

It is also possible that the address in the getelementptr is described by a constant. When this happens the memory address will be directly known. These rules operate on load and store instructions respectively.

Rule 60: IdentifyGeneralLoadBarrier

```
(defrule IdentifyGeneralLoadBarrier
         "Creates a load memory barrier hint if it turns out that the load
         instruction in question refers to a register that isn't an
         AllocaInstruction, GetElementPointerInstruction, or Constant."
         (declare (salience 3))
         (Stage Analysis $?)
         ?fct <- (Analyze ?target for load ?t0)
         (object (is-a LoadInstruction) (ID ?t0) (Parent ?p))
         (object (is-a BasicBlock) (ID ?p) (Parent ?r))
         =>
         (retract ?fct)
         (assert (instruction ?t0 memory target UNKNOWN)
                 (block ?p reads from UNKNOWN)
                 (Block ?p has a MemoryBarrier)
                 (Region ?r has a MemoryBarrier)))
```

Rule 61: IdentifyGeneralStoreBarrier

Finally, if it turns out that if address is defined by an instruction other than a getelementptr, constant, or alloca then it is necessary to mark the code as having a memory barrier. The same actions take place for both load and store instructions respectively. The only difference is one refers to reads and the other refers to writes.

Once all load and store instructions have been analyzed in this fashion it is necessary to actually update the associated instructions, basic blocks, and regions with this new information. The first six rules handle the propagation of new information destined for the ReadsFrom and WritesTo multislots of basic blocks and regions.

Rule 62: InsertIntoBlocksReadsFrom

```
(defrule InsertIntoBlockReadsFrom
        "Puts the target value into the target basic block's ReadsFrom
        multifield"
        (declare (salience -9))
        (Stage Analysis $?)
        ?fct <- (block ?p reads from ?t)
        ?bb <- (object (is-a BasicBlock) (ID ?p) (Parent ?q))
        =>
        (retract ?fct)
        (assert (Region ?q reads from ?t))
        (slot-insert$ ?bb ReadsFrom 1 ?t))
```

This rule adds the determined memory address to the ReadsFrom multislot of the target basic block. The same information is propagated to the parent of the basic block.

Rule 63: InsertIntoBlockWritesTo

This rule adds a memory address to the WritesTo multislot of the target basic block. This same information is propagated to the parent of the target basic block.

Rule 64: InsertIntoRegionReadsFrom

```
(defrule InsertIntoRegionReadsFrom
        "Puts the target value into the target basic block's ReadsFrom
        multifield"
        (declare (salience -9))
        (Stage Analysis $?)
        ?fct <- (Region ?p reads from ?t)
        ?bb <- (object (is-a Region) (ID ?p) (Parent ?q))
        (exists (object (is-a Region) (ID ?q)))
        =>
        (retract ?fct)
        (assert (Region ?q reads from ?t))
        (slot-insert$ ?bb ReadsFrom 1 ?t))
```

This rule is meant to update the target region with another element that it reads from as well as tell its parent of that information as well.

Rule 65: InsertIntoRegionReadsFrom-ParentDoesntExist

This rule handles the case where the parent of the target region does not exist. It does not propagate this information any further.

Rule 66: InsertIntoRegionWritesTo

```
(defrule InsertIntoRegionWritesTo
         "Puts the target value into the target basic block's ReadsFrom
         multifield"
         (declare (salience -9))
         (Stage Analysis $?)
         ?fct <- (Region ?p writes to ?t)
         ?bb <- (object (is-a Region) (ID ?p) (Parent ?q))
         (exists (object (is-a Region) (ID ?q)))
         =>
         (retract ?fct)
         (assert (Region ?q writes to ?t))
         (slot-insert$ ?bb WritesTo 1 ?t))
```

This rule is tasked with updating the WritesTo multifield of the target region and propagate that information to its parent.

Rule 67: InsertIntoRegionWritesTo-ParentDoesntExist

However, if the parent does not exist then it is no longer necessary to propagate that information any further.

Once the reads from and writes to information has been fully propagated through out the entire CFG it is necessary to update specific basic blocks and regions with the knowledge that they have a memory barrier.

Rule 68: UpdateBlockHasMemoryBarrier

```
(defrule UpdateBlockHasMemoryBarrier
         (declare (salience -10))
         (Stage Analysis $?)
         ?fct <- (Block ?b has a MemoryBarrier)
         ?obj <- (object (is-a BasicBlock) (ID ?b)
                         (Parent ?p)
                         (HasMemoryBarrier FALSE))
         (exists (object (is-a Region) (ID ?p)))
         =>
         (retract ?fct)
         (assert (Region ?p has a MemoryBarrier))
         (modify-instance ?obj (HasMemoryBarrier TRUE)))
```

This rule not only updates the target basic block with the knowledge of having a memory barrier but also propagates the information to its parent region.

Rule 69: RetractBlockHasMemoryBarrier

However, there is the possibility that the target basic block already is aware that it has a memory barrier. In this case, the knowledge is propagated to its parent region only.

Rule 70: UpdateRegionHasMemoryBarrier

```
(defrule UpdateRegionHasMemoryBarrier
         (declare (salience -10))
         (Stage Analysis $?)
         ?fct <- (Region ?b has a MemoryBarrier)
         ?obj <- (object (is-a Region) (ID ?b)
                         (Parent ?p)
                         (HasMemoryBarrier FALSE))
         (exists (object (is-a Region) (ID ?p)))
         =>
         (retract ?fct)
         (assert (Region ?p has a MemoryBarrier))
         (modify-instance ?obj (HasMemoryBarrier TRUE)))
```

This rule is responsible for making a region aware of the fact it has memory barrier. It will also notify its parent region as well.

Rule 71: UpdateRegionHasMemoryBarrier-ParentIsntObject

As always, this rule is meant to handle the case where the parent of the target region does not exist and as such only updates the current region.

Rule 72: RetractRegionHasMemoryBarrier

```
(defrule RetractRegionHasMemoryBarrier
         (declare (salience -10))
         (Stage Analysis $?)
         ?fct <- (Region ?b has a MemoryBarrier)
         ?obj <- (object (is-a Region) (ID ?b)
                         (Parent ?p)
                         (HasMemoryBarrier TRUE))
         (exists (object (is-a Region) (ID ?p)))
         =>
         (assert (Region ?p has a MemoryBarrier))
         (retract ?fct))
```

This rule handles the case where the target region already is aware of the fact that it has a memory barrier. As such, it only propagates this information to its parent region.

Rule 73: RetractRegionHasMemoryBarrier-ParentIsntObject

If the region does not have a parent region then only the fact associated with this action is retracted. Nothing more.

Once these rules have updated the entire function with knowledge of what it reads from and writes to it is necessary to start assigning memory targets to instructions.

Rule 74: SetMemoryTargetForInstruction

```
(defrule SetMemoryTargetForInstruction
         (declare (salience -10))
         (Stage Analysis $?)
         ?fct <- (instruction ?t0 memory target ?target)
         ?obj <- (object (is-a Instruction) (ID ?t0))
         =>
         (retract ?fct)
         (modify-instance ?obj (MemoryTarget ?target)))
```

While every instruction has a MemoryTarget field, it is only really used by load and store instructions. A value of nil within the memory target field means that the instruction does not target memory at all. This is not used by the current wavefront scheduling implementation as it really is not that useful.

Once this rule updates all corresponding instructions it is necessary to start what can only be described as extended memory analysis.

3.3.4 Extended Memory Analysis

This part of the analysis section is responsible for identifying dependencies between load and store instructions with respect to the newly identified memory targets. Most of the rules in this stage are copies of the rules defined earlier. The only difference being that the word -*Extended* has been added to the names of the rule. Because of this, the majority of this part will not be explained as the code does the same as the rules described earlier.

There are, however, three new rules that handle properly ordering loads and stores.

Rule 75: StoreToLoadDependency

```
(defrule StoreToLoadDependency
         (Stage ExtendedMemoryAnalysis $?)
         (object (is-a StoreInstruction) (Parent ?p) (ID ?t0)
                 (TimeIndex ?ti0)
                 (MemoryTarget ?sym0))
         (object (is-a LoadInstruction) (Parent ?p) (ID ?t1)
                 (TimeIndex ?ti1&:(< ?ti0 ?ti1))
                 (MemoryTarget ?sym1))
         (test (or (eq ?sym0 ?sym1) (eq ?sym0 UNKNOWN)))
         =>
         (assert (Instruction ?t1 consumes ?t0)
                 (Instruction ?t0 produces ?t1)))
```

This rule compares a store and a load to see if the load is dependent on the result of the store. This can take the form of either that the two memory targets are equal, or that the location the store instruction writes to is not known. In either case, the rule generates a producer/consumer relationship between the two instructions.

Rule 76: StoreToStoreDependency

```
(defrule StoreToStoreDependency
         (Stage ExtendedMemoryAnalysis $?)
         (object (is-a StoreInstruction) (Parent ?p) (ID ?t0)
                 (TimeIndex ?ti0)
                 (MemoryTarget ?sym0))
         (object (is-a StoreInstruction) (Parent ?p) (ID ?t1)
                 (TimeIndex ?ti1&:(< ?ti0 ?ti1))
                 (MemoryTarget ?sym1))
         (test (or (eq ?sym0 ?sym1) (eq ?sym0 UNKNOWN)))
         =>
         (assert (Instruction ?t1 consumes ?t0)
                 (Instruction ?t0 produces ?t1)))
```

This rule ensures that two store instructions occur in the proper order. Without this rule it is totally possible that the second store instruction could be scheduled to occur before the first which would be incorrect. This rule uses the same test condition as the previous rule as the same match applies. It even asserts a producer and consumer relationship as well.

Rule 77: LoadToStoreDependency

```
(defrule LoadToStoreDependency
         (Stage ExtendedMemoryAnalysis $?)
         (object (is-a LoadInstruction) (Parent ?p) (ID ?t0)
                 (TimeIndex ?ti0)
                 (MemoryTarget ?sym0))
         (object (is-a StoreInstruction) (Parent ?p) (ID ?t1)
                 (TimeIndex ?ti1&:(< ?ti0 ?ti1))
                 (MemoryTarget ?sym1))
         (test (or (eq ?sym0 ?sym1) (eq ?sym0 UNKNOWN)))
         =>
         (assert (Instruction ?t1 consumes ?t0)
                 (Instruction ?t0 produces ?t1)))
```

This rule ensures that a load occurs before a store if necessary. Failure to recognize this case can cause incorrect loads to occur which would cause the program to become incorrect. As with the previous two rules, this rule asserts a producer/consumer relationship between the instructions.

The following 10 rules, as stated earlier, are duplicates of commands described earlier. They are put here to handle the producer and consumer relationships constructed from the previous three rules.

Rule 78: MergeConsumers-Extended

Rule 79: MergeProducers-Extended

Rule 80: MergeConsumers-Multi-Extended

Rule 81: MergeProducers-Multi-Extended

Rule 82: MergeConsumers-Only-Extended

Rule 83: MergeProducers-Only-Extended

Rule 84: ExtendedInjectConsumers

Rule 85: ExtendedInjectProducers

Rule 86: SetifyInstructionProducers-Extended

Rule 87: SetifyInstructionConsumers-Extended

Rule 88: SetifyLocalDependencies-Extended

Once all of these rules have finished running, it is now possible to construct a set of paths for each region that is deemed to be a viable scheduling candidate.

3.4 Path Construction

Path construction is one of the aspects of wavefront scheduling that was described in detail within the paper. Unfortunately, it used a bit vector to represent the set of paths that flow through each basic block[4]. Emulating a bit vector in CLIPS is actually very difficult to do, and would yield little benefit. Instead, it was decided to generate a path objects which contained the list of elements, in order, that make up the path. Paths can consist of regions, loops, and basic blocks. However, only regions can be targeted for path generation at this point. Loops are disallowed because an effective method of handling backedges was not determined during implementation. As it stands right now, attempting to generate the set of paths through a loop will result in an infinite loop. Constructing a path has three aspects to it: initialization, traversal, and building.

3.4.1 Path Initialization

When starting path construction it is necessary to initialize the path generator by *pointing* it to the entrances of the target region. Generally speaking an entrance to a region can be a basic block or a region.

Rule 89: StartPathConstruction-BasicBlock

```
(defrule StartPathConstruction-BasicBlock
         (declare (salience 3))
         (Stage Path $?)
         ?r0 <- (object (is-a Region)
                        (ID ?n)
                        (Class Region)
                        (CanWavefrontSchedule TRUE)
                        (Entrances $? ?a $?))
         (object (is-a BasicBlock) (ID ?a) (Parent ?n))
         =>
         (make-instance (gensym*) of Path (Parent ?n) (Contents ?a)))
```

The first case is extremely straight forward and signifies that the basic block is directly owned by the region being pathed.

Rule 90: StartPathConstruction-NestedEntrance

```
(defrule StartPathConstruction-NestedEntrance
         (declare (salience 3))
         (Stage Path $?)
         ?r0 <- (object (is-a Region)
                        (ID ?n)
                        (Class Region)
                        (CanWavefrontSchedule TRUE)
                        (Entrances $? ?a $?)
                        (Contents $? ?z $?))
         (object (is-a Region) (ID ?z) (Parent ?n) (Entrances $? ?a $?))
         (object (is-a BasicBlock) (ID ?a) (Parent ~?n))
         =>
         (make-instance (gensym*) of Path (Parent ?n) (Contents ?z)))
```

The entrance to a given region is a sub-region if the following conditions are met:

1. The entrance block is not owned by the target region

2. The entrance block is the same for a sub region owned by the target region.

When these two conditions are met it means that the nested region becomes the the entrance to the target region.

Once paths have been created for all entrances across all viable regions it is necessary to start traversing these regions, building up the paths that represent each of them.

3.4.2 Traversing the region

Traversing a region encompasses eight different cases to be handled. These cases are:

1. Traversing from a basic block to a basic block.

2. Traversing from a basic block to a region.

3. Traversing from a region to a basic block.

4. Traversing from a region to a region.

5. Traversing from a basic block to an exit.

6. Traversing from a region to an exit.

7. Terminating a path at a basic block with no successors.

8. Terminating a path at a region with no successors.

The first four cases are meant to add elements to a given path. The remaining four are meant to mark paths as closed. A closed path is one that can no longer continue in the given region because the next element either does not exist or is not part of the target region. These eight cases take a deferred where the action to perform is determined but not immediately performed.

Rule 91: PathConstruction-BasicBlockToBasicBlock

```
(defrule PathConstruction-BasicBlockToBasicBlock
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Parent ?p) (ID ?id) (Contents $? ?curr)
                          (Closed FALSE))
         (object (is-a BasicBlock) (ID ?curr) (Parent ?p)
                 (Successors $? ?next $?))
         (object (is-a BasicBlock) (ID ?next) (Parent ?p))
         (object (is-a Region) (Class Region) (ID ?p))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Add ?next to ?id)))
```

The first case is the simplest where the given path is at a basic block and has chosen a successor that is directly owned by the target region.

Rule 92: PathConstruction-BasicBlockToRegion

```
(defrule PathConstruction-BasicBlockToRegion
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Closed FALSE) (Parent ?p) (ID ?id)
                          (Contents $? ?c))
         (object (is-a BasicBlock) (ID ?c) (Parent ?p) (Successors $? ?s $?))
         (object (is-a Region) (ID ?v) (Parent ?p) (Entrances $? ?s $?))
         (object (is-a Region) (Class Region) (ID ?p))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Add ?v to ?id)))
```

The second case occurs when the current path is at a basic block and chooses a successor that is not directly owned by the target region. It is necessary to find out which region the target block is an entrance to and select that region.

Rule 93: PathConstruction-RegionToBasicBlock

```
(defrule PathConstruction-RegionToBasicBlock
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Closed FALSE) (Parent ?p) (ID ?id)
                          (Contents $? ?c))
         (object (is-a Region) (ID ?c) (Parent ?p) (Exits $? ?e $?))
         (object (is-a BasicBlock) (ID ?e) (Parent ?p))
         (object (is-a Region) (Class Region) (ID ?p))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Add ?e to ?id)))
```

The third case occurs when the current path is at a region and the exit of that region is directly owned by the target region.

Rule 94: PathConstruction-RegionToRegion

```
(defrule PathConstruction-RegionToRegion
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Closed FALSE) (Parent ?p) (ID ?id)
                          (Contents $? ?c))
         (object (is-a Region) (ID ?c) (Parent ?p) (Exits $? ?e $?))
         (object (is-a Region) (ID ?q) (Parent ?p) (Entrances $? ?e $?))
         (object (is-a Region) (Class Region) (ID ?p))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Add ?q to ?id)))
```

The fourth case occurs when the current path is at a region and selects an exit node which is an entrance to another region owned by the target region.

Rule 95: PathConstruction-BasicBlockToExit

```
(defrule PathConstruction-BasicBlockToExit
         "Marks the current path as finished because we've reached an exit to
         the current region"
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Parent ?p) (ID ?id) (Contents $? ?curr)
                          (Closed FALSE))
         (object (is-a BasicBlock) (ID ?curr) (Parent ?p) (Successors $? ?e $
             ?))
         (object (is-a Region) (Class Region) (ID ?p) (Exits $? ?e $?))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Close ?id with ?e)))
```

The fifth case handles the situation where the current path is at a basic block and it chooses a successor that is marked as an exit of the target region.

Rule 96: PathConstruction-RegionToExit

```
(defrule PathConstruction-RegionToExit
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Closed FALSE) (Parent ?p) (ID ?id)
                          (Contents $? ?c))
         (object (is-a Region) (ID ?c) (Parent ?p) (Exits $? ?e $?))
         (object (is-a Region) (Class Region) (ID ?p) (Exits $? ?e $?))
         ; both the inner and outer regions have the same exit...thus the
         ; curent nested region is a terminator for one path
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Close ?id with ?e)))
```

The sixth case occurs when the current path is at a region and selects an exit node that is also marked as an exit of the target region.

Rule 97: PathConstruction-BlockNoExit

```
(defrule PathConstruction-BlockNoExit
         "We are at a basic block that has no successors... usually the end of
         a function"
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Parent ?p) (ID ?id) (Closed FALSE)
                          (Contents $? ?a))
         (object (is-a BasicBlock) (ID ?a) (Parent ?p) (Successors))
         (object (is-a Region) (ID ?p) (Class Region))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Close ?id with nil)))
```

The seventh case occurs when current path is at a basic block that has no successors. This usually occurs when the terminator instruction of the target block is a return or an invoke instruction.

Rule 98: PathConstruction-RegionNoExit

```
(defrule PathConstruction-RegionNoExit
         "We are at a region that doesn't have an exit...Not sure if LLVM
         allows this but let's handle it."
         (declare (salience 2))
         (Stage Path $?)
         ?path <- (object (is-a Path) (Parent ?p) (ID ?id) (Closed FALSE)
                          (Contents $? ?a))
         (object (is-a Region) (ID ?a) (Parent ?p) (Exits))
         (object (is-a Region) (ID ?p) (Class Region))
         =>
         (send ?path .IncrementReferenceCount)
         (assert (Close ?id with nil)))
```

The eighth case occurs when the current path is at a region which does not have any exits. This usually signifies that the region returns or invokes another function.

3.4.3 Updating the paths

While path traversal is important, it is also important to ensure that *all* paths through a region are independently described. This makes it necessary to not only add elements to an existing path but also make copies of said path if there is more than one choice that can be made for a given path. It is also important to close paths that have been concluded. There are four cases associated with path building.

1. Adding an element to a copy of the target path
2. Adding an element to the original instance of the target path
3. Closing a copy of the target path
4. Closing the original instance of the target path

The traversal of paths will cause facts to be asserted for a given path that are unordered but describe an action to take. The use of reference counting is done to determine if the given element should be concatenated to the original path or a copy.

There are four rules defined with each one handling a different case.

Rule 99: AddToPath-Copy

```
(defrule AddToPath-Copy
         "Makes a copy of the current path object and concatenates the symbol
         in question to the end of the list. This rule is fired when the
         reference count of the given path object is greater than one."
         (declare (salience 1))
         (Stage Path $?)
         ?fct <- (Add ?next to ?id)
         ?hint <- (object (is-a Path) (Closed FALSE) (ID ?id) (Parent ?p)
                          (ReferenceCount ?rc&:(> ?rc 1)))
         =>
         (send ?hint .DecrementReferenceCount)
         (retract ?fct)
         (make-instance (gensym*) of Path (Parent ?p)
                        (Contents (send ?hint get-Contents) ?next)))
```

The first case occurs when there are multiple paths that can be created from a given path. Each path beyond the first one has a copy of the path made and the chosen *next* element added. This ensures that the original path that this copy was spawned from is not modified in anyway.

Rule 100: AddToPath-Concat

```
(defrule AddToPath-Concat
         "Concatenates the next element of the path directly to the original
         path object. This rule fires when the reference count of the path is
         equal to one"
         (declare (salience 1))
         (Stage Path $?)
         ?fct <- (Add ?next to ?id)
         ?hint <- (object (is-a Path) (Closed FALSE) (ID ?id)
                          (ReferenceCount 1))
         =>
         (retract ?fct)
         (modify-instance ?hint (ReferenceCount 0)
                          (Contents (send ?hint get-Contents) ?next)))
```

The second case occurswhen the there is only one unsatisfied reference left. Instead of making another copy, the original path has the corresponding element concatenated to it.

Rule 101: ClosePath-Copy

```
(defrule ClosePath-Copy
         "Closes a path by making a copy of the target path"
         (declare (salience 1))
         (Stage Path $?)
         ?fct <- (Close ?id with ?bb)
         ?hint <- (object (is-a Path) (Closed FALSE) (ID ?id) (Parent ?p)
                          (ReferenceCount ?rc&:(> ?rc 1)))
         =>
         (send ?hint .DecrementReferenceCount)
         (retract ?fct)
         (make-instance (gensym*) of Path (Closed TRUE) (ExitBlock ?bb)
                        (Contents (send ?hint get-Contents))))
```

The third case closes a copy of a given path because it has been discovered that such an action should be taken. As with the first case, this case occurs when there is more than one choice for the given path.

Rule 102: ClosePath-Update

```
(defrule ClosePath-Update
         "Closes a path via an in-place update"
         (declare (salience 1))
         (Stage Path $?)
         ?fct <- (Close ?id with ?bb)
         ?hint <- (object (is-a Path) (Closed FALSE) (ID ?id)
                          (ReferenceCount 1))
         =>
         (retract ?fct)
         (modify-instance ?hint (ReferenceCount 0) (Closed TRUE)
                          (ExitBlock ?bb)))
```

The fourth case occurs when there is only one choice is left and it turns out that the path must be closed. Instead of making a copy of the path the original path is closed.

In the paper on wavefront scheduling it is noted that the bit vector can be used to provide different operations such as subset, superset, etc[4]. This capability is emulated by the next two rules.

Rule 103: AddPathToBlock

```
(defrule AddPathToBlock
         "Adds the given path name to the target block if it turns out that
         it is part of it"
         ; make sure that this happens after paths have been made
         (declare (salience 1))
         (Stage Path $?)
         (object (is-a Path) (Closed TRUE) (ID ?id) (Contents $? ?b $?))
         ?bb <- (object (is-a BasicBlock) (ID ?b))
         ;make sure that we don't have this already
         (test (eq (member$ ?i (send ?bb get-Paths)) FALSE))
         =>
         (slot-insert$ ?bb Paths 1 ?id))
```

The first one adds the name of the target path to the set of paths the given basic block is a part of.

Rule 104: AddPathToRegion

The second case is activated if it turns out that the element part of the given path is a region or loop. The body of the code is basically the same, the only difference is that the target is a Region instead of a BasicBlock.

The next rule generates the list of elements (basic block or regions) used during wavefront advancement.

Rule 105: TraversePathForElementInjection

```
(defrule TraversePathForElementInjection
         (Stage PathUpdate $?)
         (object (is-a Path) (Closed TRUE) (ID ?p) (Contents $? ?a ?b $?))
         ?o0 <- (object (is-a Diplomat) (ID ?a))
         ?o1 <- (object (is-a Diplomat) (ID ?b))
         =>
         (if (eq FALSE (member$ ?a (send ?o1 get-PreviousPathElements))) then
           (slot-insert$ ?o1 PreviousPathElements 1 ?a))
         (if (eq FALSE (member$ ?b (send ?o0 get-NextPathElements))) then
           (slot-insert$ ?o0 NextPathElements 1 ?b)))
```

Like the previous two rules, this rule takes elements out of a closed path but adds them to the Previous and Next path elements as specified in the code. The reason for doing this is to easily determine the set of elements that should be added in place of target block on the wavefront. It may turn out that there are exits or successors that are not elements of current region and it is wrong to include them in the wavefront for that region. This rule causes an increase in space consumption to occur to offset a potential blowup in time taken from potentially recomputing these elements.

Once these rules have been fired, the path construction phase is complete. At this point every single region that is a viable candidate for wavefront scheduling is aware of all its paths. It is now possible start actually applying wavefront scheduling.

3.5 Wavefront Initialization

This section was not discussed at all in the paper [4] due to the fact that it involves implementation specific details. This section is responsible for creating a wavefront for each viable region in the given function. Before the wavefront can be constructed it is necessary to determine the set of entrances for each region. These entrances can take the form of a block or region. These rules are similar to the entrance selector rules used in path construction.

Rule 106: InitializeWavefrontSchedulingForARegion-SelectBlockDirectly

```
(defrule InitializeWavefrontSchedulingForARegion-SelectBlockDirectly
         (declare (salience 2))
         (Stage WavefrontInit $?)
         (object (is-a Region) (CanWavefrontSchedule TRUE) (ID ?r)
                 (Entrances $? ?e $?))
         (object (is-a BasicBlock) (ID ?e) (Parent ?r))
         =>
         (assert (Add ?e to wavefront for ?r)))
```

If it turns out that an entrance to the given region is a basic block then a fact is asserted to that effect.

Rule 107: InitializeWavefrontSchedulingForARegion-AssertRegionInstead

```
(defrule InitializeWavefrontSchedulingForARegion-AssertRegionInstead
         (declare (salience 2))
         (Stage WavefrontInit $?)
         (object (is-a Region) (CanWavefrontSchedule TRUE) (ID ?r)
                 (Entrances $? ?e $?))
         (object (is-a BasicBlock) (ID ?e) (Parent ~?r))
         (object (is-a Region) (Parent ?r) (Entrances $? ?e $?) (ID ?q))
         =>
         (assert (Add ?q to wavefront for ?r)))
```

However, if the entrance to the region is a basic block that is not a direct child of the target region then it is necessary to figure out which directly owned subregion shares this entrance with the target region. As with the previous rule, the insertion does not take effect immediately.

Once all viable regions have had all of their entrances analyzed it is necessary to merge corresponding facts together. There are four rules defined for this act.

Rule 108: MergeWavefrontCreationContents-Convert-SingleSingle

```
(defrule MergeWavefrontCreationContents-Convert-SingleSingle
         (declare (salience 1))
         (Stage WavefrontInit $?)
         ?f0 <- (Add ?v0 to wavefront for ?r)
         ?f1 <- (Add ?v1 to wavefront for ?r)
         (test (neq ?f0 ?f1))
         =>
         (retract ?f0 ?f1)
         (assert (Create wavefront for ?r containing ?v0 ?v1)))
```

This rule is extremely similar to the merging rules found earlier in this document. It takes two facts and merges them together. This is necessary because it is completely valid to have multiple entrances for a given region [4]. While this does not happen in practice, due to the way LLVM creates regions, it is very smart to handle it in case the region selector is ever changed.

Rule 109: MergeWavefrontCreationContents-Convert-MultiSingle

```
(defrule MergeWavefrontCreationContents-Convert-MultiSingle
         (declare (salience 1))
         (Stage WavefrontInit $?)
         ?f0 <- (Add ?v0 to wavefront for ?r)
         ?f1 <- (Create wavefront for ?r containing $?g0)
         =>
         (retract ?f0 ?f1)
         (assert (Create wavefront for ?r containing $?g0 ?v0)))
```

This rule is meant to merge the odd singular fact with the plural one.

Rule 110: MergeWavefrontCreationContents-Convert-MultiMulti

```
(defrule MergeWavefrontCreationContents-Convert-MultiMulti
         (declare (salience 1))
         (Stage WavefrontInit $?)
         ?f0 <- (Create wavefront for ?r containing $?g0)
         ?f1 <- (Create wavefront for ?r containing $?g1)
         (test (neq ?f0 ?f1))
         =>
         (retract ?f0 ?f1)
         (assert (Create wavefront for ?r containing $?g0 $?g1)))
```

This rule is meant to merge the numerous create wavefront facts into a single unifying one.

Rule 111: ConvertWavefrontCreationFact

```
(defrule ConvertWavefrontCreationFact
         (declare (salience 1))
         (Stage WavefrontInit $?)
         ?f0 <- (Add ?v0 to wavefront for ?r)
         =>
         (retract ?f0)
         (assert (Create wavefront for ?r containing ?v0)))
```

This rule is meant to handle cases where there is only one entrance to the given region.

Rule 112: ConstructInitialWavefront

```
(defrule ConstructInitialWavefront
         (Stage WavefrontInit $?)
         ?f0 <- (Create wavefront for ?r containing $?w)
         =>
         (retract ?f0)
         (make-instance (gensym*) of Wavefront (Parent ?r) (Contents ?w)))
```

The next step is to create the wavefront object associated with each viable region.

3.6 Wavefront Scheduling

This section models the implementation details from the paper [4] as best as possible. The only parts that the paper actually discussed in detail was use of compensation path vectors to keep track of compensation copy generation and opening and closing blocks on the wavefront[4]. The rest of the rules associated with this section had to be inferred from an understanding of how wavefront scheduling works. To this end it was necessary to impose some extra limitations that have been hinted at throughout the paper. Some of these limitations are artificial while others are a side effect of adapting wavefront scheduling to work with superscalar architectures. The limitations are as follows:

1. Call instructions are not allowed to be scheduled out of their original block.
2. Instructions that follow a call instruction are invalid scheduling targets.
3. Instructions are not able to be scheduled into a target block if it turns out that there are memory barriers between the original block and the block on the wavefront.
4. Anything below a split block is not able to be scheduled into or above the split block.
5. Blocks that contain a return or invoke terminator instruction are deemed invalid scheduling targets.

The majority of these limitations came about as a result of limitations in the wavefront scheduling implementation itself. For example, it was determined that moving call instructions out of the original block could not only be extremely unsafe but also have potential side effects that could not be determined safely. The limitation of not being able to move memory modifying instructions above other memory modifying instructions with an unknown memory target was instituted because it was determined that performing infinitely deep pointer analysis would be a project within itself.

Preventing instructions from below a split block to be scheduled above or into the split block was a concious choice due to the fact that program profiling would be required to accurately determine which path was more likely to be taken. On Itanium based system, this speculation could be easily handled through the use of predicate registers. However, predication is quite limited within LLVM and as such it was decided that speculative moves were going to be disallowed.

The final limitation was instituted during the testing of this implementation where it was found that, sometimes, certain paths were unable to have instructions scheduled into. This causes some instructions to be scheduled earlier in some paths yet remain in the original block for others. The amount of work required to synchronize this would have been overwhelming.

While this document has already stated that the activation and firing of rules is *unordered*, it is important to mention it again because this implementation of wavefront scheduling applies itself across *all* regions before continuing to the next substage. While the rules in the previous sections followed a pretty consistent pattern the actual application of wavefront scheduling will seem to be ”all over the place”.

3.6.1 Introducing a Second Control Fact

There are so many different operations that a *second* control fact had to be defined to handle the need for sub-stages.

Rule 113: InitializeWavefrontSchedulingFacts

```
(defrule InitializeWavefrontSchedulingFacts
        (declare (salience 1001))
        (Stage WavefrontSchedule $?)
        =>
        (assert (Substage Init Identify PhiIdentify PhiNode PhiNodeUpdate
                          Pathing Strip Inject Acquire Slice AnalyzeInit
                          Analyze SliceAnalyze MergeInit Merge MergeUpdate
                          ReopenBlocks Ponder Rename DependencyAnalysis
                          ScheduleObjectCreation ScheduleObjectUsage
                          ResetScheduling InitLLVMUpdate LLVMUpdate
                          AdvanceInit AdvanceIdentify Advance AdvanceEnd
                          Update)))
```

Rule 114: NextWavefrontSchedulingSubstage

This rule operates on the substage control fact and is tasked with advancing the substages. This code is nearly identical to the rule that advances the target stage. The only difference is that it operates on sub-stages instead of stages.

Rule 115: RetractSubstageCompletely

Once all substages have been completed this rule retracts the substage fact completely.

3.6.2 Initialization Substage

Now that all of the wavefront objects have been constructed it is necessary to construct a PathAggregate object for each element (basic block or region) on the wavefront. The PathAggregate object is responsible for keeping track of information such as basic blocks that contain valid scheduling targets, compensation path vectors, and much more.

This substage will be fired once each time a given wavefront is advanced. As stated earlier the explanation of these rules will be given to the reader in terms of an example. Therefore, figure 3.1 has been determined to be the most flexible and useful example money can buy.

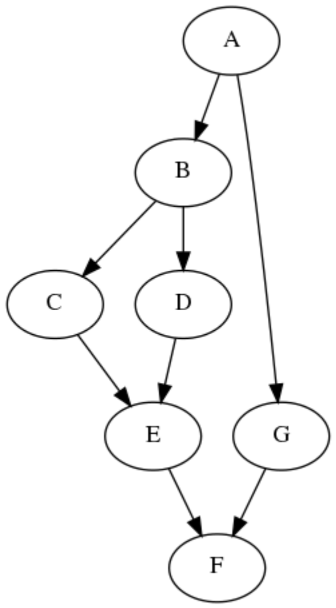

Figure 3.1: The best example money can buy.

There are two rules in this part, the difference between the following two rules mainly deals with if the element on the wavefront is a block or a region.

Rule 116: ConstructPathAggregateForBlock

```
(defrule ConstructPathAggregateForBlock
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage Init $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         ?bb <- (object (is-a BasicBlock) (ID ?e))
         =>
         (assert (Propagate aggregates of ?e))
         (bind ?stopIndex (- (length$
                              (send ?bb get-Contents)) 1))
         (make-instance (gensym*) of PathAggregate
                        (Parent ?e)
                        (OriginalStopIndex ?stopIndex)))
```

Each element (basic block or region) on the wavefront will need to have a path aggregate constructed for it. A path aggregate is an object that keeps track of a wide variety of information for the element it is attached to. Only one path aggregate is created per element on the wavefront.

Rule 117: ConstructPathAggregateForRegion

Path aggregates are constructed for regions as well, the only difference is that a region on a wavefront is not actually a valid scheduling target. Instead, the path aggregate is responsible for propagating instruction information.

Once each element on the wavefront has a path aggregate constructed, it is necessary to identify which basic blocks on the wavefront can actually be scheduled into.

3.6.3 Valid Block Identification

The paper detailing how wavefront scheduling works describes that elements *below* the wavefront are those that have yet to be scheduled into. It also describes elements *above* the wavefront as being elements that have already been scheduled and are unable to be scheduled any further. Consequently, elements that are *on* the wavefront are those that can be scheduled into[4]. Taking this definition a step further, one can define that the elements below the wavefront contain instructions that may be viable targets to be scheduled into one or more blocks currently on the wavefront. This part enforces the some of the limitations defined earlier by marking regions and split blocks that are elements on the wavefront as impossible to have instructions scheduled into. Elements that do not fall into this category will be basic blocks that are not split nodes. It is important to note that split blocks still have basic block scheduling applied to them.

Rule 118: AssertIdentifySpansInitial

```
(defrule AssertIdentifySpansInitial
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage Identify $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         =>
         (assert (Picked ?e for ?r)))
```

This rule causes each element on the wavefront to be analyzed to determine if it should have instructions scheduled into it. In Figure 3.1 on page 107, if the wavefront consisted of C, D, and G then a fact would be asserted for each one.

Rule 119: IdentifySpanSkips-SplitBlock

```
(defrule IdentifySpanSkips-SplitBlock
         (declare (salience 50))
         (Stage WavefrontSchedule $?)
         (Substage Identify $?)
         ?fct <- (Picked ?e for ?r)
         ?bb <- (object (is-a BasicBlock) (ID ?e))
         (test (send ?bb .IsSplitBlock))
         =>
         (retract ?fct)
         (assert (Schedule ?e for ?r)))
```

This rule prevents split blocks from having instructions scheduled into it. As stated earlier, this is to prevent speculation from occuring. In Figure 3.1 on page 107, this rule would fire on blocks A and B because they are both split blocks. This would cause A and B to only have basic block scheduling applied.

Rule 120: SkipRegion

Obviously, it does not make sense to attempt to schedule a region. Thus this rule causes the region to be ignored. In Figure 3.1 on page 107, this rule would apply if any of the nodes were regions instead of basic blocks.

If an element on the wavefront is neither a split block or region then the next rule applies where analysis begins on identifying paths to be analyzed for scheduling targets.

Rule 121: IdentifySpans

```
(defrule IdentifySpans
         (declare (salience 50))
         (Stage WavefrontSchedule $?)
         (Substage Identify $?)
         ?fct <- (Picked ?e for ?r)
         ?bb <- (object (is-a BasicBlock) (ID ?e) (Paths $?paths))
         (test (not (send ?bb .IsSplitBlock)))
         =>
         (retract ?fct)
         (modify-instance ?bb (IsOpen TRUE))
         (assert (Build paths for ?e from $?paths)))
```

If a basic block on the wavefront is not a split block, then it is possible to analyze the set of paths it is on. This rule will fire on blocks C,D,E, G, and F in Figure 3.1 on page 107.

Rule 122: BuildUpPaths

```
(defrule BuildUpPaths
         (declare (salience 25))
         (Stage WavefrontSchedule $?)
         (Substage Identify $?)
         ?fct <- (Build paths for ?e from ?path $?paths)
         (object (is-a Path) (ID ?path) (Contents $?c))
         =>
         (retract ?fct)
         (assert (Build paths for ?e from $?paths)
                 (Check path ?path for block ?e)))
```

This rule binds the target basic block to each path it is on. In Figure 3.1 on page 107, if C is the current target then the path A, B, C, E, F would be selected for analysis as that is the only path that C is a part of.

Rule 123: RetractPathBuildUp

Once all paths have been "built" for the target block it is necessary to retract the fact.

3.6.4 Wavefront Block Analysis

Based on the facts asserted in the previous part it is necessary to enforce the rest of the limitations described earlier. The biggest of these being call and memory barriers. A call barrier is a flag that denotes the presence of a call instruction in a given element (basic block or region). Since call barriers act as artificial cleave points, it is necessary to describe this change to CLIPS by describing elements as completely invalid or potentially valid.

An element is considered completely invalid if it follows a block or region with a call barrier. The block that contains the call barrier is considered to be potentially valid because there may be schedulable instructions that precede the first call instruction in the given block. This delineation has the side effect of reducing the "scheduling space" for blocks on the wavefront allowing the implementation to run faster while preventing the consumption of memory to represent knowledge that will never be used.

A memory barrier is a flag that denotes that a given element on a path has an unknown memory reference. Unlike call barriers, memory barriers are a little more fluid. It is possible for the barrier to be moved if the instructions that make up the barrier are scheduled.

Rule 124: DispatchDivideBlock

```
(defrule DispatchDivideBlock
         (declare (salience 200))
         (Stage WavefrontSchedule $?)
         (Substage Pathing $?)
         ?fct <- (Check path ?p for block ?e)
         (object (is-a Path) (ID ?p) (Contents $? ?e $?rest))
         (object (is-a BasicBlock) (ID ?e))
         =>
         (retract ?fct)
         (assert (Scan path ?p for block ?e with contents $?rest)))
```

This rule acts a barrier that modifies the facts asserted in rule 122. It selects all of the elements on the given path that *follow* the target basic block on the wavefront.

Rule 125: AnalyzePathElements

```
(defrule AnalyzePathElements
         (Stage WavefrontSchedule $?)
         (Substage Pathing $?)
         ?fct <- (Scan path ?p for block ?e with contents ?curr $?rest)
         ?bb <- (object (is-a BasicBlock) (ID ?curr))
         =>
         (retract ?fct)
         (if (= 0 (length$ (send ?bb get-Successors))) then
           (assert (CompletelyInvalid blocks for ?e are ?curr))
           (return))
         (if (send ?bb .IsSplitBlock) then
           (assert (CompletelyInvalid blocks for ?e are $?rest)
                   (PotentiallyValid blocks for ?e are ?curr))
           (return))
         (if (send ?bb get-HasCallBarrier) then
           (assert (CompletelyInvalid blocks for ?e are $?rest)
                   (PotentiallyValid blocks for ?e are ?curr))
           (return))
         (if (send ?bb get-HasMemoryBarrier) then
           (assert (Element ?curr has a MemoryBarrier for ?e)))
         (assert (PotentiallyValid blocks for ?e are ?curr)
                 (Scan path ?p for block ?e with contents $?rest)))
```

This rule checks a given basic block to see if it is a valid scheduling target for the target basic block on the wavefront. In Figure 3.1 on page 107, selecting C as the target would cause the path A, B, C, E, F to be analyzed as E, F. Now if E is defined as having a call barrier then it is marked as valid, F is marked as invalid, and control terminates. If it turns out that E is a split block then F would be considered invalid. If F was the end of the function then it would be marked as invalid. However, in Figure 3.1, both E and F are considered to be valid scheduling targets.

Rule 126: AnalyzePathElements-Region

```
(defrule AnalyzePathElements-Region
         (Stage WavefrontSchedule $?)
         (Substage Pathing $?)
         ?fct <- (Scan path ?p for block ?e with contents ?curr $?rest)
         ?bb <- (object (is-a Region) (ID ?curr))
         =>
         (retract ?fct)
         (if (send ?bb get-HasCallBarrier) then
           (assert (CompletelyInvalid blocks for ?e are $?rest)
                   (PotentiallyValid blocks for ?e are ?curr))
           (return))
         (if (send ?bb get-HasMemoryBarrier) then
           (assert (Element ?curr has a MemoryBarrier for ?e)))
         (assert (PotentiallyValid blocks for ?e are ?curr)
                 (Scan path ?p for block ?e with contents $?rest)))
```

This rule fires if one of the elements on the given path is a region. The only difference is that the check for the exit block is missing because it does not matter if the region exits.

Rule 127: RetractCompletedFact

This rule will only fire if it turns out that the given path section consisted entirely of potentially valid blocks.

3.6.5 Block Stripping and Simplification

Now, it is necessary to merge and strip out elements that were described as invalid on one path but valid on another.

Rule 128: MergePotentiallyValidBlocks

```
(defrule MergePotentiallyValidBlocks
         (declare (salience 2))
         (Stage WavefrontSchedule $?)
         (Substage Strip $?)
         ?pv0 <- (PotentiallyValid blocks for ?e are $?t)
         ?pv1 <- (PotentiallyValid blocks for ?e are $?q)
         (test (and (neq ?pv0 ?pv1) (subsetp ?t ?q)))
         =>
         (retract ?pv0 ?pv1)
         (assert (PotentiallyValid blocks for ?e are $?q)))
```

The actual rule follows the merge behavior that has been described several times throughout this document. The only difference is that the test also contains a check to see if one is a subset of the other.

Rule 129: MergeCompletelyInvalid

This rule does the same thing as the previous rule, the only difference is that it operates on the set of completely invalid elements instead of the potentially valid ones.

Rule 130: RetractPotentiallyValidBlocksThatAreCompletelyEnclosed

```
(defrule RetractPotentiallyValidBlocksThatAreCompletelyEnclosed
         (Stage WavefrontSchedule $?)
         (Substage Strip $?)
         (CompletelyInvalid blocks for ?e are $?t)
         ?pv1 <- (PotentiallyValid blocks for ?e are $?q)
         (test (subsetp ?q ?t))
         =>
         (retract ?pv1))
```

Sometimes it turns out that different paths will have different knowledge about the viability a given element. If it turns out that a set of elements are considered both valid and invalid, the invalid marking takes priority.

Rule 131: StripoutIndividualElementsFromPotentiallyValid

```
(defrule StripoutIndividualElementsFromPotentiallyValid
         (declare (salience -1))
         (Stage WavefrontSchedule $?)
         (Substage Strip $?)
         ?f0 <- (PotentiallyValid blocks for ?e are $?before ?car $?rest)
         (CompletelyInvalid blocks for ?e are $? ?car $?)
         =>
         (retract ?f0)
         (assert (PotentiallyValid blocks for ?e are $?before $?rest)))
```

In the case where *specific* elements are considered invalid out of a set of potentially valid, it is necessary to strip those elements out.

Rule 132: RetractEmptyPotentiallyValid

This stripping of individual elements deemed invalid can sometimes cause a potentially valid fact to not actually refer to any elements. In this case it is necessary to retract it.

Once the facts associated with target blocks on all wavefronts have been simplified and stripped it is necessary to update the associated path aggregates.

3.6.6 Block Injection

This sub stage updates the path aggregate associated with each block on the wavefront. The information the path aggregate receives is the list of completely invalid and potentially valid blocks as well as which blocks have call and memory barriers.

Rule 133: InjectPotentiallyValidBlocks

```
(defrule InjectPotentiallyValidBlocks
         (Stage WavefrontSchedule $?)
         (Substage Inject $?)
         ?fct <- (PotentiallyValid blocks for ?e are ?t $?q)
         ?pa <- (object (is-a PathAggregate) (Parent ?e))
         =>
         (retract ?fct)
         (assert (PotentiallyValid blocks for ?e are $?q))
         (if (eq FALSE (member$ ?t (send ?pa get-PotentiallyValid))) then
           (slot-insert$ ?pa PotentiallyValid 1 ?t)))
```

Obviously, this rule adds the set of elements merged in the previous part into the path aggregate associated with the target block on the wavefront. Using C in Figure 3.1 on page 107, the associated path aggregate would have E and F marked as potentially valid elements for C.

Rule 134: InjectCompletelyInvalidBlocks

```
(defrule InjectCompletelyInvalidBlocks
        (Stage WavefrontSchedule $?)
        (Substage Inject $?)
        ?fct <- (CompletelyInvalid blocks for ?e are ?t $?rest)
        ?pa <- (object (is-a PathAggregate) (Parent ?e))
        =>
        (retract ?fct)
        (assert (CompletelyInvalid blocks for ?e are $?rest))
        (if (eq FALSE (member$ ?t (send ?pa get-CompletelyInvalid))) then
          (slot-insert$ ?pa CompletelyInvalid 1 ?t)))
```

This rule is nearly identical to the previous rule. The only difference is that it updates the list of completely invalid blocks instead of potentially valid.

The next two rules are responsible for ”cleaning” up after the poentially valid and completely invalid elements of a given block on the wavefront have been injected into the target path aggregate.

Rule 135: PotentiallyValidBlocksUpdate-Last

Rule 136: CompletelyInvalidBlocksUpdate-Last

The following two rules handle notifying the target path aggregate that elements below the target block on the wavefront have memory or call barriers respectively.

Rule 137: InjectMemoryBarrierBlocks

Rule 138: InjectCallBarrierBlocks

3.6.7 Acquiring valid Compensation Path Vectors

The wavefront scheduling paper [4] describes a compensation path vector as binding a given instruction to the set of paths that the instruction exists on[4]. The compensation path vector, or CPV, is useful in keeping track of how many compensation copies of a given instruction must be generated as it is scheduled into blocks on the wavefront.

This substage takes the list of potentially valid blocks for a given block on the wavefront and extracts all instructions out of these blocks that could potentially be scheduled into the target block on the wavefront. Obviously, there are instructions that are unable to be moved due to various reasons including

- The instruction is a phi node
- The instruction is a terminator for a given block
- The instruction is a call instruction
- The instruction has a call dependency

These conditions are imposed to ensure that the set of instructions that are selected are safe to be moved. Once the list of valid instructions for all blocks in the set of potentially valid have been identified it is necessary to acquire the associated CPV. This CPV is either reloaded or created. A new CPV is created when a target instruction has never been targeted for scheduling. If the instruction already has a CPV created for it then it, is used again. This allows one CPV to represent an instruction throughout its entire scheduling lifetime.

Rule 139: SelectValidCPVs

```
(defrule SelectValidCPVs
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         (object (is-a BasicBlock) (ID ?e) (IsOpen TRUE))
         ?pa <- (object (is-a PathAggregate) (ID ?ag) (Parent ?e)
                        (PotentiallyValid $?pv))
         =>
         (assert (For ?e find CPVs for $?pv)))
```

This is a barrier rule that starts the process of CPV generation by asserting a fact requesting the act to take place.

Rule 140: FindValidCPVsForBlock

```
(defrule FindValidCPVsForBlock
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (For ?e find CPVs for ?pv $?pvs)
         (object (is-a BasicBlock) (ID ?pv) (Contents $?instructions))
         =>
         (retract ?fct)
         (assert (For ?e find CPVs for $?pvs)
                 (Get CPVs out of ?pv for ?e using $?instructions)))
```

This rule will cause the target basic block to become a CPV generator.

Rule 141: SkipRegionsForFindingValidCPVsForBlock

Obviously, it is impossible to create compensation path vectors from a region. Therefore, if one is encountered it is skipped.

Rule 142: RetractValidCPVsForBlock

The firing of this rule signifies that all given CPVs have been found for the given basic block on the wavefront.

Each basic block that has been tagged as CPV factory now has to be analyzed to determine which instructions can be scheduled and should have a compensation path vector created. The next five rules take the failure approach where instructions that match against these rules will be ignored.

Rule 143: IgnorePHIInstructions

```
(defrule IgnorePHIInstructions
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Get CPVs out of ?pv for ?e using ?inst $?insts)
         (object (is-a PhiNode) (ID ?inst))
         =>
         (retract ?fct)
         (assert (Get CPVs out of ?pv for ?e using $?insts)))
```

It is not possible to move PhiNodes and as such should be skipped if encountered.

Rule 144: DisableInstructionsDependentOnPhis

```
(defrule DisableInstructionsDependentOnPhis
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Get CPVs out of ?pv for ?e using ?inst $?insts)
         (object (is-a Instruction) (ID ?inst)
                 (DestinationRegisters $? ?reg $?))
         (object (is-a PhiNode) (ID ?reg))
         =>
         (retract ?fct)
         (assert (Get CPVs out of ?pv for ?e using $?insts)))
```

Sometimes a phi node will produce an address that a store instruction writes to. If this is the case then it is never necessary to generate a CPV for the store instruction because it is dependent on the phi node.

Rule 145: DisableInstructionsDependentOnPhis-SourceRegisters

```
(defrule DisableInstructionsDependentOnPhis-SourceRegisters
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Get CPVs out of ?pv for ?e using ?inst $?insts)
         (object (is-a Instruction) (ID ?inst) (SourceRegisters $? ?reg $?))
         (object (is-a PhiNode) (ID ?reg))
         =>
         (retract ?fct)
         (assert (Get CPVs out of ?pv for ?e using $?insts)))
```

It is also necessary to check the source registers of instructions to see if the given instruction is dependent on the result of a phi node.

Rule 146: IgnoreCallInstructions

```
(defrule IgnoreCallInstructions
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Get CPVs out of ?pv for ?e using ?inst $?insts)
         (object (is-a CallInstruction) (ID ?inst))
         =>
         (retract ?fct)
         (assert (Get CPVs out of ?pv for ?e using $?insts)))
```

Call instructions are skipped due to the limitations imposed by this implementation.

Rule 147: IgnoreTerminatorInstructions

```
(defrule IgnoreTerminatorInstructions
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Get CPVs out of ?pv for ?e using ?inst $?insts)
         (object (is-a TerminatorInstruction) (ID ?inst))
         =>
         (retract ?fct)
         (assert (Get CPVs out of ?pv for ?e using $?insts)))
```

Moving a terminator instruction makes absolutely zero sense and as such they are skipped.

Rule 148: TagValidCPVs

```
(defrule TagValidCPVs
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Get CPVs out of ?pv for ?e using ?inst $?insts)
         (object (is-a Instruction) (ID ?isnt) (IsTerminator FALSE)
                 (HasCallDependency FALSE))
         =>
         (retract ?fct)
         (assert (Get CPVs out of ?pv for ?e using $?insts)
                 (Marked ?inst as valid for block ?e)))
```

If a given instruction ”fails” to match against the previous five rules then it may be a valid to schedule it. This rule fires on an instruction if it is not a terminator instruction and does not have a call dependency. This rule marks instructions that should have a compensation path vector generated.

Rule 149: RetractDrainedGetCPVFacts

This rule handles facts asserted by rule 148 that have no more instructions to process.

Once the set of compensation path vectors has been determined for all elements on the wavefront it is necessary to determine if it is necessary to actually create a CPV for the target instruction or just reuse a pre-existing one.

Rule 150: ReloadCPVIntoNewAggregate

```
(defrule ReloadCPVIntoNewAggregate
         "Put the CPV that has already been created into the target path
         aggregate"
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Marked ?inst as valid for block ?e)
         (exists (object (is-a CompensationPathVector) (Parent ?inst)))
         (object (is-a CompensationPathVector) (Parent ?inst) (ID ?cpvID))
         ?agObj <- (object (is-a PathAggregate) (ID ?ag) (Parent ?e)
                           (ImpossibleCompensationPathVectors $?icpv))
         (object (is-a Instruction) (ID ?inst) (NonLocalDependencies $?nlds)
                 (DestinationRegisters ?reg))
         (test (eq FALSE (member$ ?cpvID $?icpv)))
        =>
         (retract ?fct)
         (if (eq FALSE (member$ ?inst (send ?agObj get-InstructionList)))
           then (slot-insert$ ?agObj InstructionList 1 ?inst))
         (if (eq FALSE (member$ ?reg (send ?agObj get-InstructionList)))
           then (slot-insert$ ?agObj InstructionList 1 ?reg))
         (foreach ?nld ?nlds
                  (bind ?instList (send ?agObj get-InstructionList))
                  (if (eq FALSE (member$ ?nld ?instList)) then
                    then (slot-insert$ ?agObj InstructionList 1 ?nld)))
         (slot-insert$ ?agObj CompensationPathVectors 1 ?cpvID))
```

If the target instruction already has a compensation path vector associated with it then there is no need to construct a new one. This rule takes that compensation path vector and updates the path aggregate of the corresponding basic block on the wavefront so that it knows that the target instruction has the potential to be scheduled into it.

Rule 151: MakeCPV

```
(defrule MakeCPV
         (Stage WavefrontSchedule $?)
         (Substage Acquire $?)
         ?fct <- (Marked ?inst as valid for block ?e)
         (not (exists (object (is-a CompensationPathVector) (Parent ?inst))))
         (object (is-a Instruction) (ID ?inst) (Parent ?pv)
                 (DestinationRegisters ?reg) (NonLocalDependencies $?nlds))
         (object (is-a BasicBlock) (ID ?pv) (Paths $?paths))
         ?pa <- (object (is-a PathAggregate) (ID ?ag) (Parent ?e))
         =>
         ; We need to disable the stores from moving when their dependencies
         (retract ?fct)
         (bind ?name (gensym*))
         (slot-insert$ ?pa CompensationPathVectors 1 ?name)
         (make-instance ?name of CompensationPathVector (Parent ?inst)
                        (Paths $?paths)
                        (OriginalBlock ?pv))
         (if (eq FALSE (member$ ?inst (send ?pa get-InstructionList)))
           then (slot-insert$ ?pa InstructionList 1 ?inst))
         (if (eq FALSE (member$ ?reg (send ?pa get-InstructionList)))
           then (slot-insert$ ?pa InstructionList 1 ?reg))
         (foreach ?nld ?nlds
                  (if (eq FALSE (member$ ?nld (send ?pa get-InstructionList)))
                    then (slot-insert$ ?pa InstructionList 1 ?nld))))
```

This rule fires when an instruction needs to have a CPV created for it. This is only done once per instruction to not only save memory but also ensure that knowledge is consistent across multiple scheduling attempts.

3.6.8 Slice Creation

Once the set of CPVs have been created for a block on the wavefront it is necessary to create slices. A slice is a section of a given path between a given basic block on the wavefront and the basic block where a given CPV originated from. These slices are created once and tied to the originating basic block.

Slice creation is not described in the wavefront scheduling paper[4], however it was determined necessary to prevent the scheduling of instructions into blocks on the wavefront that would require the generation of compensation code to bring the function back into the realm of correctness.

Before the slices can be created it is necessary to take the list of compensation path vectors and determine which ones actually *need* slices to be created. Sometimes a given path that a CPV contains does not contain the target block on the wavefront. This makes it really important to get in there and scoop up all of those cases to prevent problems.

Rule 152: SetifyInstructionList

This rule is meant to convert the list of instructions stored in a given path aggregate into a set if it is not already one.

Rule 153: GenerateInitialSliceFactsForElementsOnTheWavefront

```
(defrule GenerateInitialSliceFactsForElementsOnTheWavefront
         (Stage WavefrontSchedule $?)
         (Substage Slice $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         (object (is-a BasicBlock) (ID ?e) (IsOpen TRUE))
         (object (is-a PathAggregate) (Parent ?e)
                 (CompensationPathVectors $?cpv))
         (test (> (length$ ?cpv) 0))
         =>
         (assert (Generate slices for block ?e in ?r using $?cpv)))
```

This rule takes a basic block that is on the wavefront and starts the process of slice generation.

Rule 154: GenerateFactForSlicesFromCPV

```
(defrule GenerateFactForSlicesFromCPV
         (Stage WavefrontSchedule $?)
         (Substage Slice $?)
         ?fct <- (Generate slices for block ?e in ?r using ?cpv $?cpvs)
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i)
                 (Paths $?paths))
         (object (is-a Instruction) (ID ?i) (Parent ?b))
         =>
         (retract ?fct)
         (assert (Generate slices for block ?e in ?r using $?cpvs)
                 (Generate slices for block ?e in ?r with cpv ?cpv with stop
                           block ?b using paths $?paths)))
```

This rule creates a fact for each compensation path vector that describes the range for a given slice. It also provides the list of paths that slices need to be generated for.

Rule 155: RetractEmptySlicesCreationFact

This rule fires when every single CPV for a given block on the wavefront has been analyzed by rule 154.

Rule 156: QueryCanCreateSliceForPath

```
(defrule QueryCanCreateSliceForPath
         (Stage WavefrontSchedule $?)
         (Substage Slice $?)
         ?fct <- (Generate slices for block ?e in ?r with cpv ?cpv with stop
                           block ?b using paths ?path $?paths)
         (object (is-a Path) (ID ?path) (Contents $?z))
         (test (neq FALSE (member$ ?e ?z)))
         =>
         (retract ?fct)
         (assert (Generate slice for block ?e in ?r with cpv ?cpv with stop
                           block ?b using path ?path)
                 (Generate slices for block ?e in ?r with cpv ?cpv with stop
                           block ?b using paths $?paths)))
```

This rule will mark an path as valid for slice creation if it turns out that the block on the wavefront is part of the target path.

Rule 157: QueryCantCreateSliceForPath

This rule compliments the previous rule and handles the case where the target block on the wavefront is not part of the path in question. This can happen when the originating block is not completely dominated by the block on the wavefront. Failure to handle this case can cause an infinite loop later on.

With the list of sliceable paths in hand, it is now time to actually create the slices. There are two cases associated with slice building.

Rule 158: TryConstructNewSlice

```
(defrule TryConstructNewSlice
         (Stage WavefrontSchedule $?)
         (Substage Slice $?)
         ?fct <- (Generate slice for block ?e in ?r with cpv ?cpv with stop
                            block ?b using path ?path)
         (not (exists (object (is-a Slice) (Parent ?b) (TargetPath ?path)
                                (TargetBlock ?e)))))
         (object (is-a Path) (ID ?path) (Contents $? ?e $?slice ?b $?))
         =>
         (retract ?fct)
         (make-instance (gensym*) of Slice (Parent ?b) (TargetPath ?path)
                        (TargetBlock ?e) (Contents $?slice)))
```

The first case occurs if the slice does not exist and it is necessary to create one.

Rule 159: SliceAlreadyExists

However, if the slice already exists then the target path is skipped. This rule is extremely similar to the previous rule except that it does not actually create a new slice.

Rule 160: RemoveSliceAnalysisFact

If all paths have been evaluated for a given CPV then the corresponding fact is retracted.

3.6.9 Initialization of Compensation Path Vector Analysis

Now that slices and CPVs have been constructed it is necessary to mark these CPVs as valid targets. This sub-stage is only fired *once* per wavefront iteration and acts as an entry point to scheduling instructions into open basic blocks on the wavefront. This part was not discussed in the paper [4] and came about as a way to ensure that instructions scheduled into the blocks on the wavefront would retain their original serial dependencies.

Rule 161: InitialCPVSetupForPathAggregate

```
(defrule InitialCPVSetupForPathAggregate
         "Load all of the compensation path vectors for the given path
         aggregate into the aggregates TargetCompensationPathVectors
         multifield"
         (Stage WavefrontSchedule $?)
         (Substage AnalyzeInit $?)
         (object (is-a Wavefront) (Contents $? ?blkID $?))
         ?agObj <- (object (is-a PathAggregate) (Parent ?blkID)
                           (CompensationPathVectors $?cpvIDs))
         (test (> (length$ ?cpvIDs) 0))
         =>
         (modify-instance ?agObj (TargetCompensationPathVectors $?cpvIDs)))
```

This rule is meant to activate the process of scheduling instructions into blocks on the wavefront. It marks all of the generated CPVs as valid scheduling targets.

Rule 162: SetifyTargetCompensationPathVectors

This rule is meant to convert the list of target CPVs of a given path aggregate into a set if it is not already one.

3.6.10 Compensation Path Vector Analysis

This section describes the first part of an inner *loop* that handles scheduling instructions into blocks on the wavefront. This section was not described in the wavefront scheduling paper [4] and came about as a way to ensure that scheduled instructions retain their original ordering within a basic block on the wavefront.

The next three rules identify which CPVs are actually valid targets for scheduling.

Rule 163: SelectCPVForAnalysis

```
(defrule SelectCPVForAnalysis
         (Stage WavefrontSchedule $?)
         (Substage Analyze $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         ?bb <- (object (is-a BasicBlock) (ID ?e) (IsOpen TRUE))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e)
                           (TargetCompensationPathVectors $?cpvs))
         (test (> (length$ ?cpvs) 0))
         =>
         ;clear out the cpvs
         (modify-instance ?agObj (TargetCompensationPathVectors))
         (bind ?result (create$))
         (foreach ?cpv ?cpvs
                  (bind ?o (symbol-to-instance-name ?cpv))
                  (bind ?pp (send ?o get-Paths))
                  (bind ?determinant FALSE)
                  (foreach ?p ?pp
                           (bind ?o2 (symbol-to-instance-name ?p))
                           (bind ?c2 (send ?o2 get-Contents))
                           (if ?determinant then (break) else
                             (bind ?determinant
                                   (or ?determinant
                                       (neq FALSE (member$ ?e ?c2))))))
                  (if ?determinant then
                    (bind ?result (create$ ?result ?cpv))))
         (assert (Analyze block ?e for ?r using cpvs $?result)))
```

This rule is meant to fire on CPVs that have had copies scheduled into other blocks. Sometimes, an instruction is believed to be a valid scheduling target for the given basic block on the wavefront but it turns out that none of the paths which the instruction still needs to be scheduled into contains the target basic block. This rule acts as a barrier to the next two rules.

Rule 164: SegmentCPVsApart

```
(defrule SegmentCPVsApart
         (Stage WavefrontSchedule $?)
         (Substage Analyze $?)
         ?fct <- (Analyze block ?e for ?r using cpvs ?cpv $?cpvs)
         (object (is-a BasicBlock) (ID ?e))
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i))
         =>
         (retract ?fct)
         (assert (Analyze block ?e for ?r using cpvs $?cpvs)
                 (Analyze instruction ?i { associated cpv ?cpv } for ?e)))
```

This rule takes the list of CPVs and asserts a fact to have each CPV analyzed individually by the instruction it is bound to.

Rule 165: RetractCPVSegmentationFact

This fact retracts empty versions of the fact operated on in rule 164.

Using the facts asserted by rule 164, it is necessary to categorize each instruction as being stalled, scheduled, or disabled.

Rule 166: TargetCPVIsImpossibleToScheduleIntoTargetBlock

```
(defrule TargetCPVIsImpossibleToScheduleIntoTargetBlock
         (Stage WavefrontSchedule $?)
         (Substage Analyze $?)
         ?fct <- (Analyze instruction ?i { associated cpv ?cpv } for ?e)
         ?agObj <- (object (is-a PathAggregate) (Parent ?e)
                           (InstructionList $?il))
         (object (is-a Instruction) (ID ?i)
                 (LocalDependencies $?ld)
                 (NonLocalDependencies $?nld))
         (test (not (and (subsetp ?ld ?il)
                         (subsetp ?nld ?il))))
         =>
         (retract ?fct)
         (bind ?ind (member$ ?i ?il))
         (if (neq ?ind FALSE) then
           (slot-delete$ ?agObj InstructionList ?ind ?ind))
         (assert (Cant schedule ?cpv for ?e ever)))
```

An instruction is deemed impossible to schedule into a target basic block on the wavefront if any of its local or non local dependencies are missing from the list of instructions being scheduled. This causes the instruction to be didsabled from being scheduled into the corresponding basic block.

Rule 167: TargetCPVCantBeScheduledIntoTargetBlockYet

```
(defrule TargetCPVCantBeScheduledIntoTargetBlockYet
         (Stage WavefrontSchedule $?)
         (Substage Analyze $?)
         ?fct <- (Analyze instruction ?i { associated cpv ?cpv } for ?e)
         ?paObj <- (object (is-a PathAggregate) (Parent ?e)
                           (InstructionList $?il)
                           (ScheduledInstructions $?sched))
         (object (is-a Instruction) (ID ?i) (LocalDependencies $?ld)
                 (NonLocalDependencies $?nld))
         (test (and (not (subsetp ?ld ?sched))
                    (subsetp ?ld ?il)
                    (subsetp ?nld ?il)))
         =>
         (retract ?fct)
         (assert (Cant schedule ?cpv for ?e now)))
```

An instruction is stalled if the local dependencies of the given instruction have not been scheduled yet. An instruction can be stalled multiple times until all of its dependences have been scheduled.

Rule 168: TargetCPVNeedsToBeSliceAnalyzed

```
(defrule TargetCPVNeedsToBeSliceAnalyzed
         (Stage WavefrontSchedule $?)
         (Substage Analyze $?)
         ?fct <- (Analyze instruction ?i { associated cpv ?cpv } for ?e)
         (object (is-a PathAggregate) (Parent ?e)
                 (ScheduledInstructions $?sched))
         (object (is-a Instruction) (ID ?i) (Parent ?b)
                 (LocalDependencies $?ld))
         (test (subsetp ?ld ?sched))
         (object (is-a CompensationPathVector) (ID ?cpv) (Paths $?paths))
         (object (is-a BasicBlock) (ID ?b))
         =>
         (retract ?fct)
         (bind ?validPaths (create$))
         (foreach ?z ?paths
                  (bind ?obj (instance-name (symbol-to-instance-name ?z)))
                  (if (neq FALSE
                           (member$ ?e (send ?obj get-Contents))) then
                    (bind ?validPaths (create$ ?validPaths ?z))))
         (if (> (length$ ?validPaths) 0) then
           (assert (Pull slices for range ?e to ?b for instruction ?i {
                          associated cpv ?cpv } using paths $?validPaths))))
```

If all of the local dependencies of a given instruction are scheduled, then it is possible to schedule the instruction into the target basic block on the wavefront.

The following example shows how these three classifications are used in scheduling instructions into blocks on the wavefront.

```
b2:
   %5 = phi double [ %2, %b0 ], [ %4, %b1 ]
   %6 = shl nsw i64 %n.0, 1
   %7 = sitofp i64 %6 to double
   %8 = fdiv double 1.000000e+00, %7
   %9 = fdiv double %8, %5
```

Assume that b0 and b1 are blocks on the wavefront and that %6, %7, and %8 are valid scheduling targets. The order of scheduling is described by Table 3.3.

Target Block	Iteration	Instruction	Action
b0	0	%6	
b0	0	%7	Stall due to dependency on %6
b0	0	%8	Stall due to dependency on %7
b0	1	%7	Schedule
b0	1	%8	Stall due to dependency on %7
b0	2	%8	Schedule
b1	0	%6	Schedule
b1	0	%7	Stall due to dependency on %6
b1	0	%8	Stall due to dependency on %7
b1	1	%7	Schedule
b1	1	%8	Stall due to dependency on %7
b1	2	%8	Schedule

Table 3.3: A simple example showing the process of scheduling instructions into blocks on the wavefront

However, it necessary to first analyze the slices tied to this CPV to see if the instruction can be scheduled.

Rule 169: CreateSliceSegments

```
(defrule CreateSliceSegments
         (Stage WavefrontSchedule $?)
         (Substage Analyze $?)
         ?fct <- (Pull slices for range ?e to ?b for instruction ?i {
                        associated cpv ?cpv } using paths ?path $?paths)
         (object (is-a Slice) (Parent ?b) (TargetBlock ?e) (TargetPath ?path)
                 (ID ?s))
         =>
         (retract ?fct)
         (assert (Pull slices for range ?e to ?b for instruction ?i {
                        associated cpv ?cpv } using paths $?paths)
                 (Analyze slice ?s for ?e and cpv ?cpv)))
```

This rule binds the target CPV to a corresponding slice for the target basic block on the wavefront.

Rule 170: RetractSliceSegmentFact

This rule will fire once all slices have been bound to their corresponding CPVs for a given basic block on the wavefront.

Rule 171: MergeSliceAnalysisFacts-SingleSingle

Rule 172: ConvertSingleSliceRule

Rule 173: MergeSliceAnalysisFacts-SingleMulti

Rule 174: RetractSliceAnalysisFacts-SingleMulti

Rule 175: MergeSliceAnalysisFacts-MultiMulti

Rule 176: SetifyAnalyzeSlicesFact

These six rules perform the same actions as rules described earlier in this document. The objective is to create a single fact that describes all slices that a given CPV must check when attempting to schedule into the target basic block on the wavefront.

3.6.11 Slice Analysis

The slices that were constructed earlier are now used to determine if a given CPV is able to be moved into the target block on the wavefront. This detemrination takes a *negative* approach where attempts are made to mark the given CPV as a filure. There are five different tests that are performed on a given CPV.

Rule 177: AnalyzeSliceContentsForFailure-ProducerLowerThanTargetBlock

```
(defrule AnalyzeSliceContentsForFailure-ProducerLowerThanTargetBlock
         "Does a check to make sure that non local dependencies prevent an
         instruction from being moved upward into the target block"
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices ?s $?ss)
         (object (is-a Slice) (ID ?s) (TargetBlock ?e) (Parent ?b)
                 (Contents $? ?element $?))
         (object (ID ?element) (Produces $? ?nld $?))
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i))
         (object (is-a Instruction) (ID ?i) (DestinationRegisters ?dr)
                 (NonLocalDependencies $? ?nld $?))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         =>
         (retract ?fct)
         (bind ?ind (member ?i (send ?agObj get-InstructionList)))
         (if (neq FALSE ?ind) then
           (slot-delete$ ?agObj InstructionList ?ind ?ind))
         (assert (Cant schedule ?cpv for ?e ever)))
```

The first test checks to see if an element contained in the target slice computes a producer used by the target instruction bound to the CPV in question. If this is the case then the instruction is considered to be impossible to schedule because the scheduled instruction would not be ordered correctly.

Rule 178: AnalyzeSliceContentsForFailure-CallBarrier

```
(defrule AnalyzeSliceContentsForFailure-CallBarrier
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices ?s $?ss)
         (object (is-a Slice) (ID ?s) (TargetBlock ?e) (Parent ?b)
                 (Contents $? ?element $?))
         (object (ID ?element) (HasCallBarrier TRUE))
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i))
         (object (is-a Instruction) (ID ?i) (DestinationRegisters ?dr))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         =>
         (retract ?fct)
         (bind ?ind (member$ ?i (send ?agObj get-InstructionList)))
         (if (neq FALSE ?ind) then
           (slot-delete$ ?agObj InstructionList ?ind ?ind))
         (assert (Cant schedule ?cpv for ?e ever)))
```

The second test performs a check to see if there is an element of the given slice that contains a call barrier. It does not matter if the element is a basic block or a region. If the element in the slice contains a call barrier then it is necessary to mark the CPV as impossible to schedule.

Rule 179: SliceTargetHasMemoryBarrier

```
(defrule SliceTargetHasMemoryBarrier
         "The given slice has an element that contains a memory barrier.
         A memory barrier is only created when analysis has failed to
             ascertain
         what is being read from or written to in memory."
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices ?s $?ss)
         (object (is-a Slice) (ID ?s) (TargetBlock ?e)
                 (Parent ?b) (Contents $? ?element $?))
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i))
         (object (is-a LoadInstruction|StoreInstruction) (ID ?i)
                 (DestinationRegisters ?dr))
         (object (ID ?element) (HasMemoryBarrier TRUE))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         =>
         (retract ?fct)
         (bind ?ind (member$ ?i (send ?agObj get-InstructionList)))
         (if (neq FALSE ?ind) then
           (slot-delete$ ?agObj InstructionList ?ind ?ind))
         (assert (Cant schedule ?cpv for ?e ever)))
```

The third test blocks memory operations from being scheduled if it turns out that the slice has an element with a memory barrier in place.

Rule 180: SliceTargetDoesntHaveMemoryBarrier-ModifiesSameMemory

```
(defrule SliceTargetDoesntHaveMemoryBarrier-ModifiesSameMemory
         "The given slice has an element that contains a entry in the WritesTo
         list that is the same thing as the given load or store instruction"
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices ?s $?ss)
         (object (is-a Slice) (ID ?s) (TargetBlock ?e)
                 (Parent ?b) (Contents $? ?element $?))
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i))
         ?instruction <- (object (is-a LoadInstruction|StoreInstruction)
                                 (ID ?i) (MemoryTarget ?mt)
                                 (DestinationRegisters ?dr))
         (object (ID ?element) (HasMemoryBarrier FALSE) (HasCallBarrier FALSE)
                 (WritesTo $? ?mt $?))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         =>
         (retract ?fct)
         (bind ?ind (member$ ?i (send ?agObj get-InstructionList)))
         (if (neq FALSE ?ind) then
           (slot-delete$ ?agObj InstructionList ?ind ?ind))
         (assert (Cant schedule ?cpv for ?e ever)))
```

The forth test checks to see if a store instruction writes to the same address in memory as an element in the target slice. It is marked as impossible to schedule.

Rule 181: SliceTargetDoesntHaveMemoryBarrier-HasUnknownReference

```
(defrule SliceTargetDoesntHaveMemoryBarrier-HasUnknownReference
         "Does now allow loads or stores to be moved above the given element
         regardless of if a memory barrier exists or not. This is because
             there
         is an unknown loader element"
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices ?s $?ss)
         (object (is-a Slice) (ID ?s) (TargetBlock ?e)
                 (Parent ?cpv) (Contents $? ?element $?))
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?i))
         (object (is-a LoadInstruction|StoreInstruction) (ID ?i)
                 (Parent ?q) (DestinationRegisters ?dr))
         (object (ID ?element) (WritesTo $? UNKNOWN $?))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         =>
         (retract ?fct)
         (bind ?ind (member$ ?i (send ?agObj get-InstructionList)))
         (if (neq FALSE ?ind) then
           (slot-delete$ ?agObj InstructionList ?ind ?ind))
         (assert (Cant schedule ?cpv for ?e ever)))
```

The fifth test checks to see if there is an element in the target slice that contains an unknown memory target. If this is the case, then any load or store operations are deemed impossible to schedule into the basic block on the wavefront.

Rule 182: RetractSliceAnalysis

```
(defrule RetractSliceAnalysis
         "Retract all slice analysis if it turns out there is a failure fact"
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices $?)
         (exists (Cant schedule ?cpv for ?e ?))
         =>
         (retract ?fct))
```

This rule retracts analysis facts when another fact denoting that the target compensation path vector is not able to be scheduled.

If all of these tests fail, then the instruction is deemed able to be scheduled into the basic block on the wavefront at the current time. While this stage does not actually perform the scheduling it does start the process.

Rule 183: CanScheduleIntoBlockOnSlice

```
(defrule CanScheduleIntoBlockOnSlice
         (declare (salience -2))
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices ?s $?ss)
         =>
         (retract ?fct)
         (assert (Analyze in ?e using cpv ?cpv and slices $?ss)))
```

This rule is the hidden *sixth* case where the instruction was found to be able to be scheduled into the basic block on the wavefront by the target slice.

Rule 184: CanScheduleInstructionThisIteration

```
(defrule CanScheduleInstructionThisIteration
         (declare (salience -3))
         (Stage WavefrontSchedule $?)
         (Substage SliceAnalyze $?)
         ?fct <- (Analyze in ?e using cpv ?cpv and slices)
         =>
         (retract ?fct)
         (assert (Can schedule ?cpv for ?e)))
```

If this rule fires it means that a set of slices representing all elements between between the basic block on the wavefront and the originating one were found to be free of scheduling obstructions, allowing the corresponding instruction to be scheduled this iteration.

3.6.12 Initialization of Instruction Merging

This part takes the CPVs that were categorized in the last part and adds them to the appropriate category of the path aggregate owned by the target basic block on the wavefront.

Rule 185: AddCPVToSuccessList

```
(defrule AddCPVToSuccessList
         (Stage WavefrontSchedule $?)
         (Substage MergeInit $?)
         ?fct <- (Can schedule ?cpvID for ?blkID)
         ?agObj <- (object (is-a PathAggregate) (Parent ?blkID))
         =>
         (retract ?fct)
         (slot-insert$ ?agObj MovableCompensationPathVectors 1 ?cpvID))
```

Obviously, a CPV is considered a success if it can be scheduled this iteration. This rule marks instructions of this classification to be movable.

Rule 186: FailCPVForNow

```
(defrule FailCPVForNow
         (Stage WavefrontSchedule $?)
         (Substage MergeInit $?)
         ?fct <- (Cant schedule ?cpvID for ?blkID now)
         ?agObj <- (object (is-a PathAggregate) (Parent ?blkID))
         =>
         (retract ?fct)
         (slot-insert$ ?agObj StalledCompensationPathVectors 1 ?cpvID))
```

If a CPV needs to be stalled, then this rule will mark it as such.

Rule 187: RemoveCPVFromService

```
(defrule RemoveCPVFromService
         (Stage WavefrontSchedule $?)
         (Substage MergeInit $?)
         ?fct <- (Cant schedule ?cpvID for ?blkID ever)
         ?agObj <- (object (is-a PathAggregate) (Parent ?blkID))
         ?cpvObj <- (object (is-a CompensationPathVector) (ID ?cpvID)
                           (Parent ?i))
         =>
         (retract ?fct)
         (slot-insert$ ?cpvObj Failures 1 ?blkID)
         (slot-insert$ ?agObj ImpossibleCompensationPathVectors 1 ?cpvID))
```

This rule fires if the instruction was deemed impossible to schedule into the block on the wavefront. It marks the instruction as impossible to schedule for the given basic block.

3.6.13 Merging Instructions into Blocks on the Wavefront

Scheduling instructions into a basic block on the wavefront comes in two forms. The first form is the generation of a compensation copy for a subset of the overall number of paths the target instruction is a part of. The second form covers the final set of paths or all paths depending on the situation. The rule wise the only difference between the two is that the original block that the instruction originates has all references to the target instruction removed when all paths the instruction is a part of have been scheduled into.

In the original design of this sub-stage, the original instruction was moved into the basic block representing the final set of paths. While this is more inline with how [4] described the process of scheduling into basic blocks on the wavefront, it made it extremely difficult to replace uses of the target instruction within the original basic block in LLVM. Therefore, a copy of the target instruction is scheduled into each relevant block. It also allows the original instruction to remain in the original block as a sort of sign post so that phi node generation can know when to stop propagating and generating phi nodes and replace all uses of the original instruction with the value determined to represent all potential cases. There is only one edge case with this approach and it has to do with store instructions. It was found that deleting the original instance of a store instruction inside of LLVM could potentially also delete the type information associated with it. This would cause LLVM's module verifier to kick in describing that the type *badtype* was invalid.

Fixing this issue required that only store instructions were scheduled with the original behavior of moving the original instruction into the basic block that represents the final set of paths for the given instruction. This behavior does not cause any problems because store instructions do not actually need to be renamed.

This part is broken into two sections, the first section is tasked with identifying what kind of scheduling action should be taken for a given CPV. The second section uses the information acquired in the first to either clone or move a given instruction into a basic block on the wavefront.

Rule 188: AssertScheduleCPVIntoTargetBlock

```
(defrule AssertScheduleCPVIntoTargetBlock
         (Stage WavefrontSchedule $?)
         (Substage Merge $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         (object (is-a Diplomat) (ID ?e) (IsOpen TRUE))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e)
                           (MovableCompensationPathVectors ?cpv $?))
         ;we are going to be draining this list out
         (object (is-a CompensationPathVector) (ID ?cpv) (Parent ?inst))
         =>
         (slot-delete$ ?agObj MovableCompensationPathVectors 1 1)
         (assert (Determine schedule style for ?cpv into block ?e)))
```

This rule is meant to start the process of determining what style of scheduling is required for the given CPV.

Rule 189: ScheduleStyleForCPVIsMove

```
(defrule ScheduleStyleForCPVIsMove
        "This rule attempts to determine if the CPV should be moved into the
        given block on the wavefront. If this is true then the fact to
            perform
        this action will be asserted"
        (Stage WavefrontSchedule $?)
        (Substage Merge $?)
        ?fct <- (Determine schedule style for ?cpv into block ?e)
        (object (is-a BasicBlock) (ID ?e) (Paths $?paths))
        (object (is-a CompensationPathVector) (ID ?cpv) (Paths $?cpvPaths))
        ;the two sets are the same
        (test (equal$ ?paths ?cpvPaths))
        =>
        (retract ?fct)
        (assert (Move ?cpv into ?e)))
```

This rule is fired if it turns out that only one copy of the instruction needs to be scheduled. This means that all of the paths remaining for the compensation path vector are the same as the basic block in question.

Rule 190: ScheduleStyleForCPVIsCompensate

```
(defrule ScheduleStyleForCPVIsCompensate
        "This rule attempts to determine if the CPV should be copied into the
        given block on the wavefront. If this is true then the fact to
            perform
        this action will be asserted."
        (Stage WavefrontSchedule $?)
        (Substage Merge $?)
        ?fct <- (Determine schedule style for ?cpv into block ?e)
        (object (is-a BasicBlock) (ID ?e) (Paths $?paths))
        (object (is-a CompensationPathVector) (ID ?cpv) (Paths $?cpvPaths))
        ;there are more paths in the CPV than in the block
        (test (subsetp ?paths ?cpvPaths))
        =>
        (retract ?fct)
        (assert (Clone ?cpv into ?e)))
```

This rule is fired if it turns out that the basic block being scheduled into will only consume a subset of the overall paths this compensation path vector is a part of.

Rule 191: RemoveScheduleStyleForCPV

```
(defrule RemoveScheduleStyleForCPV
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage Merge $?)
         ?fct <- (Determine schedule style for ?cpv into block ?e)
         (object (is-a BasicBlock) (ID ?e) (Paths $?paths))
         (object (is-a CompensationPathVector) (ID ?cpv) (Paths $?cpvPaths))
         ;there are more paths in the CPV than in the block
         (test (not (subsetp ?paths ?cpvPaths)))
         =>
         (retract ?fct))
```

This rule is fired if it turns out that the basic block being scheduled into has more paths than the compensation path vector. This means that scheduling an instruction into this target basic block would be speculating that the instruction is going to be executed.

The next two rules handle either scheduling a compensation copy of a given instruction or moving the instruction into the block depending on the facts asserted by rules 189 and 190.

Rule 192: MoveInstructionIntoBlock

```
(defrule MoveInstructionIntoBlock
         "Moves the given object into bottom of the given block"
         (Stage WavefrontSchedule $?)
         (Substage Merge $?)
         ?fct <- (Move ?cpv into ?e)
         ?newBlock <- (object (is-a BasicBlock) (ID ?e)
                              (Contents $?blockBefore ?last))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         ?terminator <- (object (is-a TerminatorInstruction) (ID ?last)
                                (Pointer ?tPtr) (TimeIndex ?ti) (Parent ?e))
         ?cpvObject <- (object (is-a CompensationPathVector) (ID ?cpv)
                               (Parent ?inst))
         ?newInst <- (object (is-a Instruction) (ID ?inst) (Pointer ?nPtr)
                             (Parent ?otherBlock) (Class ?class)
                             (DestinationRegisters ?register))
         ?oldBlock <- (object (is-a BasicBlock) (ID ?otherBlock)
                              (Produces $?pBefore ?inst $?pRest)
                              (Contents $?before ?inst $?rest))
         =>
         (retract ?fct)
         (modify-instance ?terminator (TimeIndex (+ ?ti 1)))
         (slot-insert$ ?newBlock Produces 1 ?register)
         (modify-instance ?oldBlock (Contents $?before $?rest)
                          (Produces $?pBefore $?pRest))
         (modify-instance ?cpvObject (Paths))
         (bind ?a (send ?newInst get-Consumers))
         (assert (Remove evidence of ?inst from instructions ?a)
                 (Recompute block ?otherBlock))
         (if (eq StoreInstruction ?class) then
           (slot-insert$ ?agObj ScheduledInstructions 1 ?inst ?register)
           (modify-instance ?newBlock (Contents $?blockBefore ?inst ?last))
           (llvm-unlink-and-move-instruction-before ?nPtr ?tPtr)
           (slot-insert$ ?cpvObject ScheduleTargets 1 ?e ?inst)
           (slot-insert$ ?cpvObject Aliases 1 ?inst ?e)
           (slot-insert$ ?agObj ReplacementActions 1 ?inst ?inst !)
           else
           (bind ?newName (sym-cat movedinstruction. (gensym*) . ?inst))
           (slot-insert$ ?cpvObject ScheduleTargets 1 ?e ?newName)
           (slot-insert$ ?cpvObject Aliases 1 ?newName ?e)
           (slot-insert$ ?agObj ReplacementActions 1 ?inst ?newName !)
           (modify-instance ?newBlock (Contents $?blockBefore ?newName ?last))
           (bind ?newPtr (llvm-clone-instruction ?nPtr ?newName))
           ;purge the list of producers and consumers
           (duplicate-instance ?inst to ?newName (ID ?newName) (Name ?newName)
                               (Pointer ?newPtr) (Producers) (Consumers)
                               (NonLocalDependencies) (LocalDependencies)
                               (TimeIndex ?ti) (Parent ?e))
           (llvm-move-instruction-before ?newPtr ?tPtr)
           (slot-insert$ ?oldBlock UnlinkedInstructions 1 ?inst)
           (slot-insert$ ?agObj InstructionPropagation 1 ?inst ?newName ?e !)
           (slot-insert$ ?agObj ScheduledInstructions 1 ?inst)))
```

This rule is meant to move an instruction from one block to another.

Generally speaking, a copy is made of the original instruction which is moved into the target block. The only exception to this action are store instructions, which are moved due to the type erasure issue discussed earlier. other reason for the copy is to allow the original instruction to become "unlinked". This strips the instruction out of the block but keeps it active. It is used to determine when phi node generation for a given instruction should stop. The idea behind that once the block that the instruction originated from is on the wavefront it is no longer necessary to propagate the phi node information because any references to the original instruction will have been maintained up to that point.

The instruction is scheduled right before the terminator instruction in the target block. This ensures correct ordering with very little overhead. The new instruction takes the index of the terminator instruction. The index of the terminator is then incremented by one. This process also causes the list of producers, consumers, local dependencies, and non local dependencies of the clone to be nullified out so that it can be recomputed relative to the current block. The list of remaining paths that have to be satisfied through the generation of a compensation copy are removed because the scheduling of this instruction satisfies all of those paths.

Rule 193: CloneInstructionIntoBlock

```
(defrule CloneInstructionIntoBlock
         "Moves the given object into bottom of the given block"
         (Stage WavefrontSchedule $?)
         (Substage Merge $?)
         ?fct <- (Clone ?cpv into ?e)
         ?newBlock <- (object (is-a BasicBlock) (ID ?e)
                      (Contents $?blockBefore ?last))
         ?agObj <- (object (is-a PathAggregate) (Parent ?e))
         ?terminator <- (object (is-a TerminatorInstruction) (Pointer ?tPtr)
                        (ID ?last) (TimeIndex ?ti) (Parent ?e))
         ?cpvObject <- (object (is-a CompensationPathVector) (ID ?cpv)
                       (Parent ?inst)
                       (Paths $?cpvPaths))
         ?newInst <- (object (is-a Instruction) (ID ?inst) (Pointer ?nPtr)
                     (Parent ?otherBlock)
                     (DestinationRegisters ?register) (Class ?class))
         =>
         ;we also need to update all CPVs within
         (retract ?fct)
         (bind ?newName (sym-cat compensation.copy. (gensym*) . ?inst))
         (bind ?newPtr (llvm-clone-instruction ?nPtr ?newName))
         ;purge the list of producers and consumers
         (modify-instance ?terminator (TimeIndex (+ ?ti 1)))
         (duplicate-instance ?inst to ?newName (ID ?newName) (Name ?newName)
                             (Pointer ?newPtr)
                             (TimeIndex ?ti) (Parent ?e))
         (llvm-move-instruction-before ?newPtr ?tPtr)
         ;we add the original name so that we don't have to do
         ; an insane number of updates to the CPVs that follow
         ; this object
         (if (eq StoreInstruction ?class) then
           (slot-insert$ ?agObj ScheduledInstructions 1 ?inst ?register)
           else
           (slot-insert$ ?agObj InstructionPropagation 1 ?inst ?newName ?e !)
           (slot-insert$ ?agObj ScheduledInstructions 1 ?inst))
         (slot-insert$ ?newBlock Produces 1 ?register)
         (modify-instance ?newBlock (Contents $?blockBefore ?newName ?last))
         (slot-insert$ ?cpvObject ScheduleTargets 1 ?e ?newName)
         (slot-insert$ ?cpvObject Aliases 1 ?newName ?e)
         (slot-insert$ ?agObj ReplacementActions 1 ?inst ?newName !)
         (assert (Recompute block ?otherBlock)
                 (Reopen blocks from ?cpv))
         (bind ?leftOvers (create$))
         (foreach ?z ?cpvPaths
                  (bind ?cPath (symbol-to-instance-name ?z))
                  (if (eq FALSE (member$ ?e (send ?cPath get-Contents))) then
                    (bind ?leftOvers (insert$ ?leftOvers 1 ?z))))
         (modify-instance ?cpvObject (Paths ?leftOvers)))
```

This rule is meant to generate a compensation copy of a given instruction to be scheduled into a target block. This rule will fire if it is found that scheduling into the target block on the wavefront will satisfy a subset of the overall number of paths the target instruction is a part of. This rule is similar to rule 192 except that the instruction is still able to be scheduled into other blocks. This rule also computes the set of paths in the target CPV that are *not* satisfied by the generation of this compensation copy and updates the CPV to reflect which paths are left.

This section can execute multiple times within a single iteration in response to the changes made to the set of elements within a given CPV.

3.6.14 Updating Instructions that have been merged

Once instruction merging has occurred for the current iteration it is necessary to update the basic blocks below the wavefront that were polled for scheduling material.

The first task involves removing any indication that a given instruction existed within its original confines.

Rule 194: RemoveInstructionsFromProducers

```
(defrule RemoveInstructionsFromProducers
         (declare (salience 768))
         (Stage WavefrontSchedule $?)
         (Substage MergeUpdate $?)
         ?fct <- (Remove evidence of ?tInst from instructions ?inst $?insts)
         ?obj <- (object (is-a Instruction) (ID ?inst)
                          (Producers $?pb ?tInst $?pa)
                          (LocalDependencies $?ldb ?tInst $?lda))
         =>
         (retract ?fct)
         (assert (Remove evidence of ?tInst from instructions $?insts))
         (modify-instance ?obj (Producers $?pb $?pa)
                          (LocalDependencies $?ldb $?lda))
         (slot-insert$ ?obj NonLocalDependencies 1 ?tInst))
```

This rule removes all evidence that the now unlinked instruction ever existed within the original basic block.

Rule 195: RetractRemoveInstructionsFromProducers

This rule terminates the removal of evidence for a given set of instructions from its original basic block.

The other task that has to be performed requires the memory targets of the basic blocks modified by merging to be recomputed.

Rule 196: StartRecomputeBlock

```
(defrule StartRecomputeBlock
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage MergeUpdate $?)
         ?fct <- (Recompute block ?b)
         ?bb <- (object (is-a BasicBlock) (ID ?b)
                        (Contents $?instructions ?last $?))
         (object (is-a TerminatorInstruction) (ID ?last))
         =>
         (retract ?fct)
         (modify-instance ?bb (ReadsFrom) (WritesTo) (HasMemoryBarrier FALSE))
         (assert (Recompute block ?b with instructions $?instructions)))
```

This rule fires in response to a fact asserted anytime an instruction is scheduled into a basic block on the wavefront. It clears out all knowledge of what the basic block reads from, writes to, and if it has a memory barrier. It is then forced to recompute this information to ensure that all information is properly updated.

Rule 197: RecomputeLoadInstructionForBlock

```
(defrule RecomputeLoadInstructionForBlock
         (declare (salience 99))
         (Stage WavefrontSchedule $?)
         (Substage MergeUpdate $?)
         ?fct <- (Recompute block ?b with instructions ?inst $?rest)
         (object (is-a LoadInstruction) (ID ?inst) (Parent ?b)
                 (MemoryTarget ?mt))
         ?bb <- (object (is-a BasicBlock) (ID ?b))
         =>
         (if (eq FALSE (member$ ?mt (send ?bb get-ReadsFrom))) then
           (slot-insert$ ?bb ReadsFrom 1 ?mt))
         (retract ?fct)
         (assert (Recompute block ?b with instructions $?rest)))
```

This rule is meant to recompute the list of memory addresses that are read from for a given basic block.

Rule 198: RecomputeStoreInstructionForBlock

```
(defrule RecomputeStoreInstructionForBlock
         (declare (salience 99))
         (Stage WavefrontSchedule $?)
         (Substage MergeUpdate $?)
         ?fct <- (Recompute block ?b with instructions ?inst $?rest)
         (object (is-a StoreInstruction) (ID ?inst) (Parent ?b)
                 (MemoryTarget ?mt))
         ?bb <- (object (is-a BasicBlock) (ID ?b))
         =>
         (if (eq FALSE (member$ ?mt (send ?bb get-WritesTo))) then
           (slot-insert$ ?bb WritesTo 1 ?mt))
         (retract ?fct)
         (assert (Recompute block ?b with instructions $?rest)))
```

This rule is meant to recompute the list of memory addresses that are written to for a given basic block.

Rule 199: RecomputeNonMemoryInstructionForBlock

```
(defrule RecomputeNonMemoryInstructionForBlock
         (declare (salience 99))
         (Stage WavefrontSchedule $?)
         (Substage MergeUpdate $?)
         ?fct <- (Recompute block ?b with instructions ?inst $?rest)
         (object (is-a BasicBlock) (ID ?b))
         (object (is-a Instruction&~LoadInstruction&~StoreInstruction)
                          (ID ?inst) (Parent ?b))
         =>
         (retract ?fct)
         (assert (Recompute block ?b with instructions $?rest)))
```

This rule handles the case where the instruction being analyzed is neither a load or store instruction. When this is the case the instruction is skipped as it has nothing to contribute.

Rule 200: FinishRecomputationForBlock

```
(defrule FinishRecomputationForBlock
         (declare (salience 98))
         (Stage WavefrontSchedule $?)
         (Substage MergeUpdate $?)
         ?fct <- (Recompute block ?b with instructions)
         ?bb <- (object (is-a BasicBlock) (ID ?b) (ReadsFrom $?rf)
                        (WritesTo $?wt))
         =>
         (retract ?fct)
         (if (or (neq FALSE (member$ UNKNOWN ?rf))
                 (neq FALSE (member$ UNKNOWN ?wt))) then
           (modify-instance ?bb (HasMemoryBarrier TRUE))))
```

This rule finishes computation for the given block by checking to see the block should be marked as having a memory barrier or not. This check is if there is a memory target that is read from or written to that is marked as UNKNOWN.

3.6.15 Opening and Closing Blocks on the Wavefront

Despite being mentioned in [4], the initial implementation of wavefront scheduling did not handle opening and closing basic blocks on the wavefront. This omission was due in part to the fact that it did not seem to be necessary. However, during testing it was found that there are cases where basic blocks need to be reopened in order to allow instructions that were previously marked impossible to schedule as able to be scheduled. This is best described using the fragment of a CFG defined in Figure 3.2 on the next page.

There is a load instruction I in basic block H that is scheduable but has a dependency on a store instruction J in basic block G that is also scheduable. This makes it necessary to schedule J first before I can be scheduled. Unfortunately, I will be considered impossible to schedule until J has been scheduled into all paths it is a part of.

In this example, J can be scheduled into A, E, and F in that order. It is important to note that I will be immediately scheduled into F because J is no longer a part of G. However, since A and E have already been marked closed it is impossible to schedule into them. Fixing this issue requires that A and E are reopened so that I can be scheduled into them.

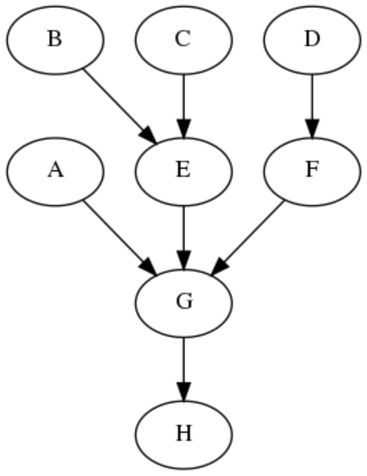

Figure 3.2: A fragment of a given function that may require blocks to be reopened on the wavefront.

On a more generic level, a basic block is able to be reopened to correct any defects in its current scheduling knowledge. A basic block is able to be opened and closed multiple times while on the wavefront [4].

Rule 201: AssertReopenBlocksOnWavefront

```
(defrule AssertReopenBlocksOnWavefront
         (Stage WavefrontSchedule $?)
         (Substage ReopenBlocks $?)
         ?fct <- (Reopen blocks from ?cpv)
         ?obj <- (object (is-a CompensationPathVector) (ID ?cpv)
                         (Failures $?failures))
         =>
         (retract ?fct)
         (modify-instance ?obj (Failures))
         (assert (From ?cpv reopen $?failures)))
```

This rule is meant to get the process of reopening blocks on the wavefront started. It takes all of the instructions that were marked as failures and asserts a fact that says these failures need to be reassessed to see if any of them are now possible to schedule.

Rule 202: ReopenBlockOnWavefront

```
(defrule ReopenBlockOnWavefront
         (Stage WavefrontSchedule $?)
         (Substage ReopenBlocks $?)
         ?fct <- (From ?cpv reopen ?fail $?failures)
         ?wave <- (object (is-a Wavefront) (ID ?w) (Closed $?a ?fail $?b)
                          (Contents $?cnts))
         ?bb <- (object (is-a BasicBlock) (ID ?fail) (IsOpen FALSE))
         ?pa <- (object (is-a PathAggregate) (Parent ?fail)
                        (ImpossibleCompensationPathVectors $?icpv))
         =>
         (modify-instance ?bb (IsOpen TRUE))
         (modify-instance ?pa (ImpossibleCompensationPathVectors)
                          (TargetCompensationPathVectors $?icpv))
         (foreach ?q ?icpv
          (slot-insert$ ?pa InstructionList 1
           (send (symbol-to-instance-name ?q) get-Parent)))
         (modify-instance ?wave (Contents $?cnts ?fail) (Closed ?a ?b))
         (retract ?fct)
         (assert (From ?cpv reopen $?failures)))
```

This rule reopens a basic block on the wavefront by taking all instructions that were deemed impossible and marking them as possible. The basic block is also marked as open once again.

Rule 203: ReadFailureToCPV

```
(defrule ReadFailureToCPV
         (Stage WavefrontSchedule $?)
         (Substage ReopenBlocks $?)
         ?fct <- (From ?cpv reopen ?fail $?failures)
         ?wave <- (object (is-a Wavefront) (ID ?w) (Closed $?c))
         (test (eq FALSE (member$ ?fail $?c)))
         ?obj <- (object (is-a CompensationPathVector) (ID ?cpv))
         =>
         (slot-insert$ ?obj Failures 1 ?fail)
         (retract ?fct)
         (assert (From ?cpv reopen $?failures)))
```

If the target basic block slated for reopening is not actually a closed basic block on the wavefront then it is skipped.

Rule 204: RetractEmptyReopenFact

This rule terminates reopening basic blocks on the wavefront for a given CPV because there are no more elements to reopen.

3.6.16 Pondering the Next Action to Take

Scheduling instructions into a basic block on the wavefront is not a one pass event. It requires restarting the scheduling process to allow stalled CPVs to get a chance to be scheduled. Unfortunately, the paper [4] does not discuss this because it is up to the implementer to figure out how to do this. This part is tasked with figuring out if it is necessary to restart the process of scheduling instructions into basic blocks on the wavefront.

Rule 205: PonderMovementIteration

```
(defrule PonderMovementIteration
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage Ponder $?)
         (object (is-a Wavefront) (ID ?r) (Contents $? ?e $?))
         ?ag <- (object (is-a PathAggregate) (Parent ?e) (ID ?pa))
         (test (> (length$ (send ?ag get-StalledCompensationPathVectors)) 0))
         =>
         (bind ?container (send ?ag get-StalledCompensationPathVectors))
         (modify-instance ?ag (StalledCompensationPathVectors)
                              (TargetCompensationPathVectors ?container)))
```

The first rule marks any CPVs previously marked as stalled as being potentially scheduable again. This spans across all wavefronts in all regions in the given function being scheduled.

Rule 206: DetermineIfAnotherMovementIsRequired

```
(defrule DetermineIfAnotherMovementIsRequired
         (declare (salience -100))
         (Stage WavefrontSchedule $?)
         ?ponder <- (Substage Ponder $?rest)
         =>
         ;this returns a tuple
         (if (any-instancep ((?inst PathAggregate))
                            (> (length$ ?inst:TargetCompensationPathVectors)
                                        0))
           then
           (retract ?ponder)
           (assert (Substage Analyze SliceAnalyze MergeInit Merge MergeUpdate
                             ReopenBlocks Ponder $?rest))
           else
           (bind ?instances (find-all-instances ((?wave Wavefront)) TRUE))
           (foreach ?instance ?instances
                    (bind ?children (send ?instance get-Contents))
                    (foreach ?child ?children
                             (bind ?obj (symbol-to-instance-name ?child))
                             (modify-instance ?obj (IsOpen FALSE))))))
```

If there are any path aggregates that contain more instructions to schedule then it is necessary to restart the instruction scheduling process.

3.6.17 Renaming Operands

After instructions have been scheduled into a target basic block on the wavefront, it is necessary to replace references to the original instruction with the compensation copy. This replacement takes place in both CLIPS and LLVM. This part was not discussed in [4] because it is a side effect of how scheduling instructions into basic blocks on the wavefront was implemented.

Rule 207: AssertReplacementActions

```
(defrule AssertReplacementActions
         "Iterates through the replacement actions multifield and asserts
         facts related to the replacement of given values with another value"
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage Rename $?)
         (object (is-a PathAggregate) (Parent ?e)
                 (ReplacementActions $? ?orig ?new ! $?))
         =>
         (assert (Replace uses of ?orig with ?new for block ?e)))
```

This rule uses the replacement actions defined during the scheduling of instructions into basic blocks on the wavefront to assert a fact starting the operand replacement process.

Rule 208: ReplaceUses

```
(defrule ReplaceUses
         (declare (salience 20))
         (Stage WavefrontSchedule $?)
         (Substage Rename $?)
         ?fct <- (Replace uses of ?orig with ?new for block ?e)
         (object (is-a Instruction) (ID ?orig) (Pointer ?oPtr))
         (object (is-a Instruction) (ID ?new) (Pointer ?nPtr))
         (object (is-a BasicBlock) (ID ?e) (Contents $? ?new $?rest))
         =>
         (retract ?fct)
         (bind ?ptrList (create$))
         (bind ?symList (create$))
         (foreach ?var $?rest
                  (bind ?obj (symbol-to-instance-name ?var))
                  (bind ?ptrList (create$ ?ptrList (send ?obj get-Pointer)))
                  (bind ?symList (create$ ?symList ?var)))
         (assert (Replace uses of symbol ?orig with symbol ?new for
                          instructions ?symList)
                 (Replace uses of pointer ?oPtr with pointer ?nPtr for
                          instructions ?ptrList)))
```

This rule identifies replacement actions that need to be taken for the block where the new instruction is found. A list of pointers and symbols is created that represents the CLIPS and LLVM representations of the instructions following the new version of the target instruction.

Rule 209: ReplaceUsesInLLVM

```
(defrule ReplaceUsesInLLVM
         (declare (salience -1))
         (Stage WavefrontSchedule $?)
         (Substage Rename $?)
         ?fct <- (Replace uses of pointer ?from with pointer ?to for
                          instructions $?p2)
         =>
         (llvm-replace-uses ?from ?to ?p2)
         (retract ?fct))
```

This rule updates LLVM, using the list of pointers defined in rule 208, such that all uses of the old instruction are replaced with the new one.

Rule 210: ReplaceUsesInCLIPS

```
(defrule ReplaceUsesInCLIPS
         (declare (salience -1))
         (Stage WavefrontSchedule $?)
         (Substage Rename $?)
         ?fct <- (Replace uses of symbol ?from with symbol ?to for
             instructions
                          ?symbol $?rest)
         ?inst <- (object (is-a Instruction) (ID ?symbol) (Operands $?operands
             ))
         =>
         (modify-instance ?inst (Operands) (SourceRegisters))
         (retract ?fct)
         (assert (Replace uses of symbol ?from with symbol ?to for instruction
                          ?symbol with operands $?operands)
                 (Replace uses of symbol ?from with symbol ?to for
                     instructions
                          $?rest)))
```

This rule is tasked with using the list of symbols found in rule 208 to update the CLIPS representation of the given instructions. Unlike LLVM replacement, this rule operates on a single instruction at a time. The list of operands for the target instruction is cleared to allow it to be cleanly recomputed.

Rule 211: ReplaceUsesInCLIPS-End

Once all instructions on the CLIPS side have been updated with the new instruction, the fact describing the action will be empty and must be retracted.

The next three rules are tasked with recomputing the set of operands for each instruction that was changed by the previous rules.

Rule 212: ReplaceIndividualInstructionUses-Match

```
(defrule ReplaceIndividualInstructionUses-Match
         (declare (salience -2))
         (Stage WavefrontSchedule $?)
         (Substage Rename $?)
         ?fct <- (Replace uses of symbol ?f with symbol ?t for instruction ?s
                          with operands ?op $?ops)
         ?inst <- (object (is-a Instruction) (ID ?s))
         (test (eq ?op ?f))
         =>
         (slot-insert$ ?inst Operands 1 ?t)
         (slot-insert$ ?inst SourceRegisters 1 ?t)
         (retract ?fct)
         (assert (Replace uses of symbol ?f with symbol ?t for instruction ?s
                          with operands $?ops)))
```

If the current operand matches the replacement target then it is necessary to add the replacement value to the operand list instead.

Rule 213: ReplaceIndividualInstructionUses-NoMatch

If the operand is not a valid replacement target then it is necessary to just add it back into the list of operands of the target instruction.

Rule 214: ReplaceIndividualInstructionUses-Empty

Once the recomputation process is complete for a given instruction it is necessary to retract the fact associated with it.

3.6.18 Updating Block Dependencies after Merging

Once an instruction has been scheduled into a basic block on the wavefront, it is necessary to update the instruction's dependencies to reflect its new environment.

Rule 215: CreateDependencyAnalysisTargets

```
(defrule CreateDependencyAnalysisTargets
         (declare (salience 10))
         (Stage WavefrontSchedule $?)
         (Substage DependencyAnalysis $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         (object (is-a BasicBlock) (ID ?e))
         (object (is-a PathAggregate) (Parent ?e) (OriginalStopIndex ?si))
         =>
         ;only look at instructions starting at the original stop index. This
         ;prevents unncessary recomputation
         (assert (Evaluate ?e for dependencies starting at ?si)))
```

This rule initializes the act of recomputing dependencies for a given basic block after scheduling has occurred.

Rule 216: MarkNonLocalDependencies

```
(defrule MarkNonLocalDependencies
         (Stage WavefrontSchedule $?)
         (Substage DependencyAnalysis $?)
         (Evaluate ?p for dependencies starting at ?si)
         ?inst <- (object (is-a Instruction) (Parent ?p)
                          (TimeIndex ?t&:(>= ?t ?si))
                          (Operands $? ?o $?))
         (object (is-a Instruction) (ID ?o) (Parent ~?p))
         =>
         (slot-insert$ ?inst NonLocalDependencies 1 ?o))
```

Now that the instruction has been moved into a new basic block it is necessary to determine which operands are actually non local dependencies.

The next 11 rules are meant to recompute dependencies for the given basic block.

Rule 217: IdentifyWAR-Wavefront

Rule 218: IdentifyRAW-Wavefront

Rule 219: IdentifyWAW-Wavefront

Rule 220: MarkCallInstructionDependency-ModifiesMemory-Wavefront

Rule 221: MarkCallInstructionDepedency-InlineAsm-Wavefront

Rule 222: MarkCallInstructionDependency-SideEffects-Wavefront

Rule 223: Wavefront-MarkHasCallDependency

Rule 224: InjectConsumers-Wavefront

Rule 225: InjectProducers-Wavefront

Rule 226: StoreToLoadDependency-Wavefront

Rule 227: StoreToStoreDependency-Wavefront

Rule 228: LoadToStoreDependency-Wavefront

These rules are quite similar to their Analysis counterparts due to the fact that only the newly added instructions are valid comparison targets. Each instruction in the corresponding basic block is only compared against set of instructions that were just scheduled. This is done to not only reduce the overall running time of the optimization, but also ensure that only knowledge generated by these rules corresponds to new information.

Rule 229: FinishedDependencyAnalysis

```
(defrule FinishedDependencyAnalysis
         (declare (salience -800))
         (Stage WavefrontSchedule $?)
         (Substage DependencyAnalysis $?)
         ?fct <- (Evaluate ?p for dependencies starting at ?)
         (object (is-a BasicBlock) (ID ?p) (Parent ?r))
         =>
         (assert (Schedule ?p for ?r))
         (retract ?fct))
```

If this rule fires then the corresponding basic block is a valid candidate for basic block scheduling.

3.6.19 Schedule Construction

Before advancing the wavefront it is necessary to apply basic block scheduling to each basic block on the wavefront. The entire process of schedule construction is as follows:

1. Take all instructions from the current basic block that have all associated dependencies already scheduled.

2. Put those instructions into an instruction group.

3. Mark those instructions as being scheduled.

4. Repeat this process with the list of unscheduled instructions until all instructions have been scheduled

The instruction groups represent not only the given time index of the target instructions, but also the fact that these instructions can be safely executed in parallel. It should be noted that this part is not mentioned in [4] because it is up to the reader to realize it is necessary.

The first part of schedule construction requires figuring out how the basic block on the wavefront is designed. There are four cases that must be handled.

Rule 230: ConstructScheduleObjectForBlock-HasPhis

```
(defrule ConstructScheduleObjectForBlock-HasPhis
         (declare (salience 10))
         (Stage WavefrontSchedule $?)
         (Substage ScheduleObjectCreation $?)
         ?fct <- (Schedule ?e for ?r)
         (object (is-a TerminatorInstruction) (Parent ?e) (ID ?last))
         (object (is-a BasicBlock) (ID ?e) (Parent ?r)
                 (Contents $? ?lastPhi ?firstNonPhi $?instructions ?last $?))
         (object (is-a PhiNode) (ID ?lastPhi))
         (object (is-a Instruction&~PhiNode&~TerminatorInstruction)
                 (ID ?firstNonPhi))
         =>
         (retract ?fct)
         (assert (Update style for ?e is ?lastPhi))
         (make-instance (gensym*) of Schedule (Parent ?e)
                        (Contents ?firstNonPhi ?instructions)))
```

The first case occurs when the basic block in question has phi nodes. Obviously, phi nodes must be skipped so it is necessary to find the first *non* phi node in the basic block. The section between the last phi node and the terminator instruction becomes the part of the basic block to schedule.

Rule 231: ConstructScheduleObjectForBlock-DoesntHavePhis

The second case occurs when the basic block in question does not have any phi nodes. When this is the case, all instructions except the terminator are marked for scheduling.

Rule 232: ConstructScheduleObjectForBlock-TerminatorOnly

The third case occurs when the basic block only contains a terminator instruction. It is skipped because there is nothing to do.

Rule 233: ConstructScheduleObjectForBlock-TerminatorAndPhisOnly

The forth case occurs when the basic block in question only contains phi nodes and a terminator instruction. The basic block is skipped because there is nothing to schedule.

Rule 234: PreschedulePhiNodes

Rule 235: PrescheduleNonLocals

These two rules pre schedule phi nodes and non locals respectively. This is done to simplify the process of basic block scheduling.

Rule 236: AssertCreateInstructionGroup

```
(defrule AssertCreateInstructionGroup
         (declare (salience -100))
         (Stage WavefrontSchedule $?)
         (Substage ScheduleObjectCreation $?)
         (object (is-a Schedule) (ID ?q))
         =>
         (assert (Create InstructionGroup for ?q)))
```

Once basic block scheduling is ready to begin, it is necessary to define the need for an initial instruction group. This rule defines such a need but does not carry it out.

3.6.20 Applying Basic Block Scheduling

This part represents a *single* iteration of the application of basic block scheduling. This part is not described in [4].

Rule 237: MakeInstructionGroupForSchedulePhase

```
(defrule MakeInstructionGroupForSchedulePhase
         (declare (salience 2701))
         (Stage WavefrontSchedule $?)
         (Substage ScheduleObjectUsage $?)
         ?fct <- (Create InstructionGroup for ?q)
         ?sched <- (object (is-a Schedule) (ID ?q) (TimeGenerator ?tg)
                           (Parent ?p) (Groups $?groups))
         (not (exists (object (is-a InstructionGroup) (TimeIndex ?tg)
                              (Parent ?p))))
         =>
         (retract ?fct)
         (bind ?name (gensym*))
         (assert (Perform Schedule ?q for ?p))
         (make-instance ?name of InstructionGroup (Parent ?p) (TimeIndex ?tg))
         (modify-instance ?sched (TimeGenerator (+ 1 ?tg))
                          (Groups ?groups ?name)))
```

The first step is to construct an instruction group that will be associated with the current time instance of the target basic block on the wavefront.

Rule 238: CanScheduleInstructionNow

```
(defrule CanScheduleInstructionNow
         (declare (salience 343))
         (Stage WavefrontSchedule $?)
         (Substage ScheduleObjectUsage $?)
         (Perform Schedule ?n for ?b)
         ?sched <- (object (is-a Schedule) (ID ?n) (Parent ?b) (Scheduled $?s)
                           (Contents ?curr $?rest))
         (object (is-a Instruction) (ID ?curr) (LocalDependencies $?p))
         (test (subsetp $?p $?s))
         =>
         (slot-delete$ ?sched Contents 1 1)
         (slot-insert$ ?sched Success 1 ?curr))
```

Once an instruction group has been created, it needs to be populated with all the instructions of the basic block that have no unscheduled dependencies.

Rule 239: MustStallInstructionForSchedule

```
(defrule MustStallInstructionForSchedule
         (declare (salience 343))
         (Stage WavefrontSchedule $?)
         (Substage ScheduleObjectUsage $?)
         (Peform Schedule ?n for ?b)
         ?sched <- (object (is-a Schedule) (ID ?n) (Scheduled $?s)
                           (Contents ?curr $?rest))
         (object (is-a Instruction) (ID ?curr) (LocalDependencies $?p))
         (test (not (subsetp $?p $?s)))
         =>
         (slot-delete$ ?sched Contents 1 1)
         (slot-insert$ ?sched Failure 1 ?curr))
```

Instructions that are unable to be scheduled this iteration are marked as failures.

Rule 240: EndInstructionScheduleAttempt

This part is finished once all instructions across all schedules have been classified as being able to be scheduled or stalled for this time index.

3.6.21 Determining if it is necessary to continue the application of basic block scheduling

This part is responsible for determining if it is necessary to restart basic block scheduling. Obviously, like the previous two parts, this part is not mentioned in [4].

Rule 241: ShouldCreateNewInstructionGroup

```
(defrule ShouldCreateNewInstructionGroup
         (declare (salience 360))
         (Stage WavefrontSchedule $?)
         (Substage ResetScheduling $?)
         (object (is-a Schedule) (ID ?n) (Success $?len) (Failure $?fail)
                 (TimeGenerator ?tg))
         (test (and (> (length$ ?fail) 0) (> (length$ ?len) 0)))
         =>
         (assert (Create InstructionGroup for ?n)))
```

First, it is necessary to check and see if more scheduling must occur for any basic block on a wavefront. This will need a new instruction group to be created to hold the instructions destined for the next time index.

Rule 242: PutSuccessfulInstructionIntoInstructionGroup

```
(defrule PutSuccessfulInstructionIntoInstructionGroup
         (declare (salience 270))
         (Stage WavefrontSchedule $?)
         (Substage ResetScheduling $?)
         ?sched <- (object (is-a Schedule) (ID ?n) (Parent ?p)
                           (Success ?targ $?) (Groups $? ?last))
         ?ig <- (object (is-a InstructionGroup) (ID ?last) (Parent ?p)
                        (TimeIndex ?t))
         (object (is-a Instruction) (ID ?targ))
         =>
         (slot-insert$ ?sched Scheduled 1 ?targ)
         (slot-delete$ ?sched Success 1 1)
         (slot-insert$ ?ig Contents 1 ?targ))
```

It is necessary to mark the successful instructions as scheduled and the failed instructions as potential scheduling targets for the next iteration.

Rule 243: PutSuccessfulStoreInstructionIntoInstructionGroup

```
(defrule PutSuccessfulStoreInstructionIntoInstructionGroup
         (declare (salience 271))
         (Stage WavefrontSchedule $?)
         (Substage ResetScheduling $?)
         ?sched <- (object (is-a Schedule) (ID ?n) (Parent ?p)
                           (Success ?targ $?) (Groups $? ?last))
         ?ig <- (object (is-a InstructionGroup) (ID ?last) (Parent ?p))
         (object (is-a StoreInstruction) (ID ?targ)
                 (DestinationRegisters ?reg))
         =>
         ;we need to schedule the target register in as well
         (slot-insert$ ?sched Scheduled 1 ?targ ?reg)
         (slot-delete$ ?sched Success 1 1)
         (slot-insert$ ?ig Contents 1 ?targ))
```

Store instructions must be handled separately because their name does not match the destination register.

Rule 244: FinishedPopulatingInstructionGroup-AssertReset

```
(defrule FinishedPopulatingInstructionGroup-AssertReset
         (declare (salience 200))
         (Stage WavefrontSchedule $?)
         (Substage ResetScheduling $?)
         (object (is-a Schedule) (ID ?n) (Parent ?p) (Contents) (Success)
          (Failure $?elements))
         (test (> (length$ ?elements) 0))
         =>
         (assert (Reset schedule ?n)))
```

Then it is necessary to determine if there is more scheduling to be done. This requirement is defined as having one or more elements determined to be unscheduable at this time index.

Rule 245: ResetTargetScheduleForAnotherGo

```
(defrule ResetTargetScheduleForAnotherGo
         (declare (salience 180))
         (Stage WavefrontSchedule $?)
         (Substage ResetScheduling $?)
         ?fct <- (Reset schedule ?n)
         ?sched <- (object (is-a Schedule) (ID ?n) (Parent ?p) (Contents)
                           (Failure $?elements))
         =>
         (retract ?fct)
         (modify-instance ?sched (Contents ?elements) (Failure))
         (assert (Reset scheduling process)))
```

All instructions marked as failures this iteration are marked as potentials for the next one.

Rule 246: ResetSchedulingProcess

```
(defrule ResetSchedulingProcess
         (declare (salience -10))
         (Stage WavefrontSchedule $?)
         ?f0 <- (Substage ResetScheduling $?rest)
         ?f1 <- (Reset scheduling process)
         =>
         (retract ?f0 ?f1)
         (assert (Substage ScheduleObjectUsage ResetScheduling $?rest)))
```

The process then restarts. This continues as long as there is a basic block that requires further scheduling.

Rule 247: FinishedPopulatingInstructionGroup-AssertFinished

```
(defrule FinishedPopulatingInstructionGroup-AssertFinished
         (declare (salience 200))
         (Stage WavefrontSchedule $?)
         (Substage ResetScheduling $?)
         (object (is-a Schedule) (ID ?n)
                 (Parent ?p) (Contents) (Success) (Failure))
         =>
         (assert (Schedule ?p using ?n in llvm)))
```

However, if all elements on all wavefronts have been completely scheduled then it is necessary to notify LLVM of the changes.

3.6.22 LLVM Update Stage Initialization

The instruction groups created for each basic block on the wavefront are now used to update the structure of these basic blocks to reflect the changes made by scheduling. This process occurs in both CLIPS and LLVM.

Rule 248: SetupSchedulingContainer

```
(defrule SetupSchedulingContainer
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage InitLLVMUpdate $?)
         (Schedule ?p using ?n in llvm)
         (object (is-a BasicBlock) (ID ?p) (Contents $? ?last))
         (object (is-a TerminatorInstruction) (ID ?last) (Pointer ?tPtr))
         =>
         (bind ?name (gensym*))
         (bind ?n2 (gensym*))
         (make-instance ?n2 of Hint (Type SymbolContainer) (Parent ?p))
         (make-instance ?name of Hint (Type Container) (Parent ?p)
                        (Point ?tPtr))
         (assert (Merge ?p at 0 using ?name and ?n2)))
```

These two environments are updated through different means. The CLIPS side is updated through symbolic replacement. The LLVM side is given pointers of the corresponding objects to manipulate. Since mixing this knowledge would be quite useless it is necessary to construct two containers. One to store pointers and another to store the symbolic names.

Rule 249: ConstructLLVMEncoding

```
(defrule ConstructLLVMEncoding
         (Stage WavefrontSchedule $?)
         (Substage InitLLVMUpdate $?)
         (Schedule ?p using ?n in llvm)
         ?fct <- (Merge ?p at ?index using ?name and ?n2)
         (object (is-a Schedule) (Parent ?p) (TimeGenerator ?ti&:(< ?index ?ti
            )))
         ?container <- (object (is-a Hint) (ID ?name) (Type Container)
                               (Parent ?p))
         ?sContainer <- (object (is-a Hint) (ID ?n2) (Type SymbolContainer)
                                (Parent ?p))
         (object (is-a InstructionGroup) (TimeIndex ?index) (Parent ?p)
                 (Contents $?contents))
         ;we need to get the elements out correctly...so we drain them out
         =>
         (retract ?fct)
         (bind ?ind (+ ?index 1))
         (assert (Merge ?p at ?ind using ?name and ?n2))
         (bind ?ptrs (symbol-to-pointer-list ?contents))
         (modify-instance ?sContainer
                          (Contents (send ?sContainer get-Contents) ?contents)
                              )
         (modify-instance ?container
                          (Contents (send ?container get-Contents) ?ptrs)))
```

Each instruction group is selected and the elements it holds are converted into a list of names and pointers to store in the proper container. This process continues until each container contains a full list of the elements that make up the current basic block.

3.6.23 Updating the Basic Block Contents in LLVM

Using the containers created in the previous part, it is necessary to use the information to update the order of instructions within each basic block on the wavefront.

Rule 250: FinishLLVMEncoding-HasPhi

```
(defrule FinishLLVMEncoding-HasPhi
         (declare (salience -12))
         (Stage WavefrontSchedule $?)
         (Substage LLVMUpdate $?)
         (Schedule ?p using ?n in llvm)
         ?f2 <- (Update style for ?p is ?lastPhi)
         ?f1 <- (Merge ?p at ?index using ?name and ?n2)
         (object (is-a Schedule) (Parent ?p) (TimeGenerator ?ti&:(>= ?index ?
             ti)))
         ?hint <- (object (is-a Hint) (ID ?name) (Type Container) (Parent ?p)
                          (Point ?tPtr) (Contents $?contents))
         ?hint2 <- (object (is-a Hint) (ID ?n2) (Type SymbolContainer)
                           (Parent ?p) (Contents $?symbols))
         ?bb <- (object (is-a BasicBlock) (ID ?p)
                        (Contents $?before ?lastPhi $?instructions ?last $?rst
                           ))
         (object (is-a TerminatorInstruction) (ID ?last) (Pointer ?tPtr))
         =>
         (modify-instance ?bb
                          (Contents $?before ?lastPhi ?symbols ?last $?rst))
         (llvm-schedule-block ?tPtr ?contents)
         (retract ?f1 ?f2)
         (unmake-instance ?hint ?hint2))
```

This rule updates basic blocks that contain phi nodes in both LLVM and CLIPS.

Rule 251: FinishLLVMEncoding-NoPhi

Where as this rule handles the basic blocks that do not contain phi nodes. The main difference between these two rules deals with where instructions are added on the CLIPS side of things.

The next three rules are responsible for cleaning up all of the facts generated by the act of basic block scheduling.

Rule 252: RetractMergeHints

Rule 253: RetractUpdateStyleHint

Rule 254: RetractScheduleHint

Once clean up has been completed it is possible to advance the wavefront. However, there is a section of wavefront scheduling that needs to be discussed first.

3.6.24 Propagating Instruction changes across the Region

When instructions have been scheduled into basic blocks on the wavefront it is necessary to keep track of the new name of that scheduled copy as the wavefront advances. This part is tasked with propagating this information to basic blocks and regions below the wavefront. The overall objective is to allow the convergence of these copies into a single value by the time the original basic block has been reached. This convergence manifests itself through the generation of phi nodes for join blocks that are below basic blocks that have different compensation copies of the same original instruction. The explanation of this stage was delayed until the end of this part to ensure that the rules made complete sense as they rely on the instruction merging actions of the previous iteration of wavefront scheduling. This part was not discussed in [4] because it is an LLVM specific action that must be taken.

Rule 255: PropagateAggregateInformation

```
(defrule PropagateAggregateInformation
         "Pulls instruction propagation information from all elements on paths
         that immediately precede this element on the wavefront and merges it
         into the path aggregate itself"
         (Stage WavefrontSchedule $?)
         (Substage Identify $?)
         (Propagate aggregates of ?e)
         ;if this element is on the wavefront then we can be certain that all
         ;of its predecessors are above it. That is the definition of being
         ;on the wavefront
         ?pa <- (object (is-a PathAggregate) (Parent ?e) (ID ?pp))
         (object (is-a Diplomat) (ID ?e) (PreviousPathElements $? ?z $?))
         (object (is-a PathAggregate) (Parent ?z)
                 (InstructionPropagation $? ?targ ?alias ? ! $?))
         =>
         ;replace parent blocks of previous path elements with the name of the
         ;element this was acquired from
         (slot-insert$ ?pa InstructionPropagation 1 ?targ ?alias ?z !))
```

This rule transfers information from the previous path elements of an element (region or basic block) on the wavefront to the element itself.

Rule 256: RetractAggregationInformation

The firing of this rule means that the corresponding element on the wavefront has been updated with all of the instruction propagation information from its previous path elements.

3.6.25 Identifying Phi Nodes

This part takes the list of propagated instructions and checks to see if it is necessary to create a phi node in the current basic block on the wavefront to converge multiple paths of execution.

Rule 257: AssertPhiNodePropagationPredicateIsBlock

```
(defrule AssertPhiNodePropagationPredicateIsBlock
         (declare (salience 1))
         (Stage WavefrontSchedule $?)
         (Substage PhiIdentify $?)
         (object (is-a Wavefront) (Parent ?r) (Contents $? ?e $?))
         ?pa <- (object (is-a PathAggregate) (Parent ?e)
                        (InstructionPropagation ?targ ?alias ?pred ! $?rest))
         =>
         (modify-instance ?pa (InstructionPropagation $?rest))
         (assert (Propagation target ?targ with alias ?alias
                              from block ?pred for block ?e)))
```

The first step is to prepare the set of propgated instructions for analysis.

3.6.26 Phi Node Construction

With the facts asserted in the last part it is now possible to determine if phi nodes need to be built to converge multiple values.

Rule 258: RemoveDuplicateElements

```
(defrule RemoveDuplicateElements
         (Stage WavefrontSchedule $?)
         (Substage PhiNode $?)
         ?f0 <- (Propagation target ?t with alias ?a from block ?p0 for
                             block ?b)
         ?f1 <- (Propagation target ?t with alias ?a from block ?p1 for
                             block ?b)
         (test (and (neq ?f0 ?f1) (neq ?p0 ?p1)))
         ?pa <- (object (is-a PathAggregate) (Parent ?b))
         =>
         (retract ?f0 ?f1)
         (slot-insert$ ?pa InstructionPropagation 1 ?t ?a ?b !))
```

If it is found that two basic blocks reference the same instruction then it is necessary to replace them with the current element and the target instruction.

Rule 259: MergePhiNodePropagationWithOtherPropagation

```
(defrule MergePhiNodePropagationWithOtherPropagation
         (Stage WavefrontSchedule $?)
         (Substage PhiNode $?)
         ?f0 <- (Propagation target ?t with alias ?a0 from block ?p0 for
                              block ?b)
         ?f1 <- (Propagation target ?t with alias ?a1 from block ?p1 for
                              block ?b)
         (test (neq ?f0 ?f1))
         =>
         (retract ?f0 ?f1)
         (assert (Create phinode targeting instruction ?t for block ?b
                          consisting of ?a0 ?p0 ?a1 ?p1 )))
```

However, if it is found that there are two basic blocks that reference the same original instruction name with different aliases then it is necessary to construct a phi node to converge the value.

Rule 260: MergePhiNodePropagationWithCreateStatement

```
(defrule MergePhiNodePropagationWithCreateStatement
         (Stage WavefrontSchedule $?)
         (Substage PhiNode $?)
         ?f0 <- (Propagation target ?t with alias ?a0 from block ?p0 for
                              block ?b)
         ?f1 <- (Create phinode targeting instruction ?t for block ?b
                         consisting of $?targets)
         =>
         (retract ?f0 ?f1)
         (assert (Create phinode targeting instruction ?t for block ?b
                          consisting of $?targets ?a0 ?p0 )))
```

Once two facts have been merged it is necessary to merge the rest of the paths into the fact as well.

Rule 261: PutUnfulfilledItemsBackIntoPropagationList

```
(defrule PutUnfulfilledItemsBackIntoPropagationList
         (declare (salience -10))
         (Stage WavefrontSchedule $?)
         (Substage PhiNode $?)
         ?f0 <- (Propagation target ?t with alias ?a0 from block ?p0 for
                              block ?b)
         ?pa <- (object (is-a PathAggregate) (Parent ?b))
         =>
         (retract ?f0)
         (slot-insert$ ?pa InstructionPropagation 1 ?t ?a0 ?b !))
```

If the target instruction propagation was never merged then it is necessary to modify it so that the target element is the current element in question.

The next two rules are responsible for creating phi nodes.

Rule 262: NamePhiNodeFromCreateStatement-NotOriginalBlock

```
(defrule NamePhiNodeFromCreateStatement-NotOriginalBlock
         (declare (salience -12))
         (Stage WavefrontSchedule $?)
         (Substage PhiNode $?)
         ?fct <- (Create phinode targeting instruction ?t for block ?b
                         consisting of $?elements)
         ?agObj <- (object (is-a PathAggregate) (Parent ?b))
         ?bb <- (object (is-a BasicBlock) (ID ?b) (Contents ?first $?rest))
         (test (eq FALSE (member$ ?t (send ?bb get-UnlinkedInstructions))))
         (object (is-a Instruction) (ID ?first) (Pointer ?bPtr))
         (object (is-a Instruction) (ID ?t) (Type ?ty))
         (object (is-a LLVMType) (ID ?ty) (Pointer ?dataType))
         =>
         (retract ?fct)
         (bind ?name (sym-cat phi. (gensym*) . ?t))
         (bind ?count (/ (length$ $?elements) 2))
         (bind ?pointers (symbol-to-pointer-list ?elements))
         (make-instance ?name of PhiNode (Parent ?b)
                        (TimeIndex 0)
                        (Pointer (llvm-make-phi-node ?name ?dataType ?count
                                                     ?bPtr ?pointers))
                        (IncomingValueCount ?count)
                        (Operands $?elements))
         ;we've scheduled the given original instruction into this block
         ; although it's just a ruse
         (slot-insert$ ?agObj ScheduledInstructions 1 ?t)
         (slot-insert$ ?agObj InstructionPropagation 1 ?t ?name ?b !)
         (slot-insert$ ?agObj ReplacementActions 1 ?t ?name !)
         (slot-insert$ ?bb Contents 1 ?name)
         (assert (Update duration for block ?b)))
```

The first rule handles the case where the basic block the phi node is being created in is *not* the original block. In this case a new phi node is created and inserted into the target basic block. This new phi node replaces all of the instruction propagation entries that went into making the phi node.

Rule 263: NamePhiNodeFromCreateStatement-OriginalBlock

```
(defrule NamePhiNodeFromCreateStatement-OriginalBlock
         (declare (salience -12))
         (Stage WavefrontSchedule $?)
         (Substage PhiNode $?)
         ?fct <- (Create phinode targeting instruction ?t for block ?b
                         consisting of $?elements)
         ?agObj <- (object (is-a PathAggregate) (Parent ?b))
         ?bb <- (object (is-a BasicBlock) (ID ?b) (Contents ?first $?rest))
         (test (neq FALSE (member$ ?t (send ?bb get-UnlinkedInstructions))))
         (object (is-a Instruction) (ID ?first) (Pointer ?bPtr))
         ?tObj <- (object (is-a Instruction) (ID ?t) (Type ?ty)
                          (Pointer ?tPtr))
         (object (is-a LLVMType) (ID ?ty) (Pointer ?dataType))
         =>
         (retract ?fct)
         (bind ?name (sym-cat phi. (gensym*) . ?t))
         (bind ?count (/ (length$ $?elements) 2))
         (bind ?pointers (symbol-to-pointer-list ?elements))
         (bind ?phiPointer (llvm-make-phi-node ?name ?dataType ?count ?bPtr
                                               ?pointers))
         (bind ?phiObj (make-instance ?name of PhiNode
                                      (Parent ?b)
                                      (TimeIndex 0)
                                      (Pointer ?phiPointer)
                                      (IncomingValueCount ?count)
                                      (Operands $?elements)))
         (llvm-replace-all-uses
           (send (symbol-to-instance-name ?t) get-Pointer)
           (send (symbol-to-instance-name ?name) get-Pointer))
         (llvm-unlink-and-delete-instruction ?tPtr)
         (unmake-instance ?tObj)
         (slot-insert$ ?agObj ScheduledInstructions 1 ?t)
         (slot-insert$ ?bb Contents 1 ?name)
         (assert (Update duration for block ?b)))
```

However, if it turns out that the target basic block is the original block then it is necessary to also delete the original instruction instance and replace all uses of the original instruction with the phi node. There is no need to propagate this information any further.

3.6.27 Phi Node Block Update

The last step after introducing phi nodes into basic blocks is to recompute the time indexes of each affected block.

Rule 264: ReindexBasicBlock

```
(defrule ReindexBasicBlock
         (Stage WavefrontSchedule $?)
         (Substage PhiNodeUpdate $?)
         ?fct <- (Update duration for block ?b)
         (object (is-a BasicBlock) (ID ?b) (Contents $?c))
         =>
         (bind ?index 0)
         (foreach ?t ?c
                  (bind ?obj (instance-address (symbol-to-instance-name ?t)))
                  (modify-instance ?obj (TimeIndex ?index))
                  (bind ?index (+ ?index 1)))
         (retract ?fct))
```

Once that is finished the act of scheduling instructions into blocks on the current wavefront may proceed.

With this part finished it is possible to discuss advancing the wavefront.

3.7 Advancing the Wavefront

Advancing the wavefront is actually made up of several steps. However, these steps are meant to simulate the algorithm provided in [4]. For the sake of brevity, this algorithm has been provided again in Figure 3.3 on the following page.

The next three rules represent identifying which elements should remain on the wavefront.

```
$delNodes$ = Set of *closed* nodes on the wavefront
**for** each node $n$ in $delNodes$ **do**
  **if** $n$ has exact one successor $s$ **then**
    **for** each predecessor $p$ of $s$ **do**
      **if** $p \notin delNodes$ **then**
        $delNodes = delNodes - \{n\}$
        break
      **end if**
    **end for**
  **end if**
**end for**
**for** each node $n$ in $delNodes$ **do**
  $Wavefront = Wavefront - \{n\}$
  $Wavefront = Wavefront \cup SuccessorsOf(n)$
**end for**
```

Figure 3.3: The algorithm for advancing the wavefront.

Rule 265: MoveContentsToDeleteNodes

```
(defrule MoveContentsToDeleteNodes
         "Moves blocks out of the contents into the closed list"
         (declare (salience 2701))
         (Stage WavefrontSchedule $?)
         (Substage AdvanceInit $?)
         ?wave <- (object (is-a Wavefront) (ID ?z) (Parent ?r)
                          (Contents $?c) (Closed $?cl))
         (test (or (> (length$ ?c) 0) (> (length$ ?cl) 0)))
         =>
         (slot-insert$ ?wave DeleteNodes 1 ?c ?cl))
```

First, the union of the set of open and closed elements (region or basic block) become the set of delete nodes for a given wavefront. A wavefront is considered completed if it does not contain any open or closed elements.

Rule 266: MarkShouldStayOnWavefront

```
(defrule MarkShouldStayOnWavefront
         (declare (salience 343))
         (Stage WavefrontSchedule $?)
         (Substage AdvanceIdentify $?)
         ?wave <- (object (is-a Wavefront) (ID ?q) (Parent ?r)
                          (DeleteNodes $?a ?b $?c))
         ?bb <- (object (is-a Diplomat) (ID ?b) (NextPathElements ?s))
         (object (is-a Diplomat) (ID ?s) (PreviousPathElements $?ppe))
         (test (not (subsetp ?ppe (send ?wave get-DeleteNodes))))
         ?agObj <- (object (is-a PathAggregate) (Parent ?b))
         =>
         (if (eq FALSE (member$ ?b (send ?wave get-Closed))) then
           (bind ?ind (member$ ?b (send ?wave get-Contents)))
           (slot-delete$ ?wave Contents ?ind ?ind)
           (slot-insert$ ?wave Closed 1 ?b))
         (modify-instance ?wave (DeleteNodes $?a $?c)))
```

The second step is to mark which elements should stay on the wavefront. An element will stay on the wavefront if it has a single successor and the successor's predecessors are not all on the wavefront[4].

Rule 267: DeleteElementFromWavefront

```
(defrule DeleteElementFromWavefront
         (declare (salience 180))
         (Stage WavefrontSchedule $?)
         (Substage Advance $?)
         ?wave <- (object (is-a Wavefront) (ID ?id) (Parent ?r)
                          (DeleteNodes ?a $?))
         (object (is-a Diplomat) (ID ?a) (NextPathElements $?npe))
         =>
         (bind ?ind (member$ ?a (send ?wave get-Contents)))
         (bind ?ind2 (member$ ?a (send ?wave get-Closed)))
         (slot-delete$ ?wave DeleteNodes 1 1)
         (if (neq ?ind FALSE) then
           (slot-delete$ ?wave Contents ?ind ?ind))
         (if (neq ?ind2 FALSE) then
           (slot-delete$ ?wave Closed ?ind2 ?ind2))
         (assert (Add into ?id blocks $?npe)))
```

Any elements left in the set of delete nodes are deleted from the wavefront and the next path elements of each deleted node is marked for insertion.

The next two rules handle advancing the wavefront.

Rule 268: PutSuccessorsOntoWavefront-Match

```
(defrule PutSuccessorsOntoWavefront-Match
         (declare (salience 100))
         (Stage WavefrontSchedule $?)
         (Substage AdvanceEnd $?)
         ?fct <- (Add into ?id blocks ?next $?rest)
         ?wave <- (object (is-a Wavefront) (ID ?id))
         =>
         (retract ?fct)
         (assert (Add into ?id blocks $?rest))
         (if (eq FALSE (member$ ?next (send ?wave get-Contents))) then
           (slot-insert$ ?wave Contents 1 ?next)))
```

This rule takes the set of elements marked for insertion by the previous rule and adds them to the wavefront.

Rule 269: PutSuccessorsOntoWavefront-NoMoreElements

The firing of this rule means that a given wavefront has been advanced.

Rule 270: PonderRestartOfWavefrontScheduling

```
(defrule PonderRestartOfWavefrontScheduling
         (declare (salience -512))
         (Stage WavefrontSchedule $?)
         ?fct <- (Substage Update $?)
         =>
         (bind ?instances (find-all-instances ((?wave Wavefront))
                                              (> (length$ ?wave:Contents) 0)))
         (if (> (length$ ?instances) 0) then
           (retract ?fct)
           (assert (Substage Init Identify PhiIdentify PhiNode
                             PhiNodeUpdate Pathing Strip Inject
                             Acquire Slice AnalyzeInit Analyze
                             SliceAnalyze MergeInit Merge MergeUpdate
                             ReopenBlocks Ponder Rename DependencyAnalysis
                             ScheduleObjectCreation ScheduleObjectUsage
                             ResetScheduling InitLLVMUpdate LLVMUpdate
                             AdvanceInit AdvanceIdentify
                             Advance AdvanceEnd Update))))
```

Once the advancement is complete it is necessary to determine if wavefront scheduling must be restarted. Wavefront scheduling does not need to be restarted if every wavefront in the target function has been completed.

Chapter 4

Testing and Performance Evaluation

4.1 Introduction

Testing the efficacy of a compiler optimization is extremely important. It is even more important to test the optimization on a machine that is some what older to determine not only what kinds of changes in code quality occur but also how long it takes to apply. The system chosen to test wavefront scheduling is a laptop with the following specifications:

- Intel Core 2 Duo T9500 (2.5 Ghz, 6 MB L2 cache, Dual core)
- 8 Gb DDR2 800
- Gentoo Linux amd64 running kernel 3.0.6

This machine is a perfect target for testing because it is over four years old and is well supported in both LLVM and GCC. This level of support also makes it more obvious if there are any improvements in code quality as a result of wavefront scheduling. Two different applications were tested to determine how much of an improvement, if any, wavefront scheduling provided. These applications are

1. CLIPS

2. FLOPS

These programs were compiled with four different compilers. These sets were: GCC, clang, opt, opt with wavefront scheduling. The opt tool comes with llvm but is quite separate from clang which is a C/C++/Objective-C frontend to LLVM. Opt takes in LLVM bitcode and applies the optimizations requested. It is also a modular tools that allows code modules to be loaded and tested. This makes debugging a new optimization not only easy but trivial as only the module has to be recompiled instead of the entire suite to reflect changes. It is with opt that wavefront scheduling is able to be applied. Acquiring the bitcode necessary for testing required the use of clang and its *-emit-llvm* option which generates LLVM bitcode instead of native machine code. This bitcode is then linked together with the rest of the components of the program to build a bitcode module representative of the entire program. Clang is also asked to *not* apply any optimizations what so-ever to the bitcode before saving it to a file. This allows the opt tool to apply the optimizations as needed.

Testing consists of the generation of 18 different versions of a given program with the differences between them being described by table 4.1.

4.2 Verification

Verification and validation will consist primarily of ensuring that the implementation of wavefront scheduling will not incorrectly compile legal code and correctly compile illegal code. Doing this requires using a series of diverse programs described in the previous section. While validating and verifying each program completely is not feasible, it is possible to use portions of each program to verify and validate the correctness of the wavefront scheduling implementation. These portions will consist of the most frequently executed functions in each program as it is here where any defects in the wavefront scheduling implementation will be immediately evident during manual analysis and program execution. Considering the implementation correct means that the code it generates will execute correctly.

For FLOPS, the program is correct if the million floating point operations per second (MFLOPS) result for each test is a positive value as well as logical. The concept of a logical

Compiler	Version	Optimization Level
opt with wavefront scheduling	LLVM 3.0 Debug	-O0
opt with wavefront scheduling	LLVM 3.0 Debug	-O1
opt with wavefront scheduling	LLVM 3.0 Debug	-O2
opt with wavefront scheduling	LLVM 3.0 Debug	-O3
opt with wavefront scheduling	LLVM 3.0 Debug	link time (LTO)
opt without wavefront scheduling	LLVM 3.0 Debug	-O0
opt without wavefront scheduling	LLVM 3.0 Debug	-O1
opt without wavefront scheduling	LLVM 3.0 Debug	-O2
opt without wavefront scheduling	LLVM 3.0 Debug	-O3
opt without wavefront scheduling	LLVM 3.0 Debug	LTO
clang	2.9 Release	-O0
clang	2.9 Release	-O1
clang	2.9 Release	-O2
clang	2.9 Release	-O3
gcc	4.5.3	-O0
gcc	4.5.3	-O1
gcc	4.5.3	-O2
gcc	4.5.3	-O3

Table 4.1: A table describing the different compilers used in testing complete with optimizations and versions

value is acquired by running FLOPS without wavefront scheduling and checking to see what is returned. For instance, if one of the MFLOPS values is 2000 for the given test computer then getting a value of 1000000 with wavefront scheduling applied usually means something is wrong. Fortunately, a case like this did not occur while verifying wavefront scheduling with FLOPS. The results of which will be discussed in the next section.

Testing CLIPS for correctness is a little more complicated due to the extensive nature of the program. Instead of exhaustively testing everything, a real expert system was used to test CLIPS to make sure that the following aspects worked correctly.

1. Fact creation
2. Defining classes
3. Defining functions
4. Creating object instances
5. Passing messages to objects

6. Defining rules

7. Pattern matching on facts and instances

8. Loading files

9. Printing out to the console.

If all of these actions worked as expected in the wavefront scheduled version of CLIPS then it would be safe to say that the wavefront scheduler is correct. The program selected to test these requirements was a basic block scheduler of Itanium assembler using all of the above criteria to schedule. The output of this program is the provided assembler rearranged into instruction groups. This output can be compared against the results of the versions of CLIPS compiled with gcc and clang. Any differences in the output represent an issue with the wavefront scheduler. While the actual results will be discussed in the next section, the wavefront scheduled versions of CLIPS always generated correct code.

With both programs working the same as their unscheduled counterparts it is safe to say that the implementation of wavefront scheduling has been verified as working. However, this does not mean that the optimization will work with all programs and there are several known caveats that are described later in this document.

4.3 Analysis of Test Results

The two programs used to verify the wavefront scheduling implementation were also used to determine if it improved performance on a superscalar processor. Each program is analyzed in two ways:

1. Scheduling time compared to compilation time

2. Execution time of the program.

4.3.1 CLIPS

Figure 4.1 on the next page compares the running time of applying wavefront scheduling to an unoptimized version of CLIPS with compiling it from source using gcc and clang at different optimization levels. This massive overhead shows that wavefront scheduling

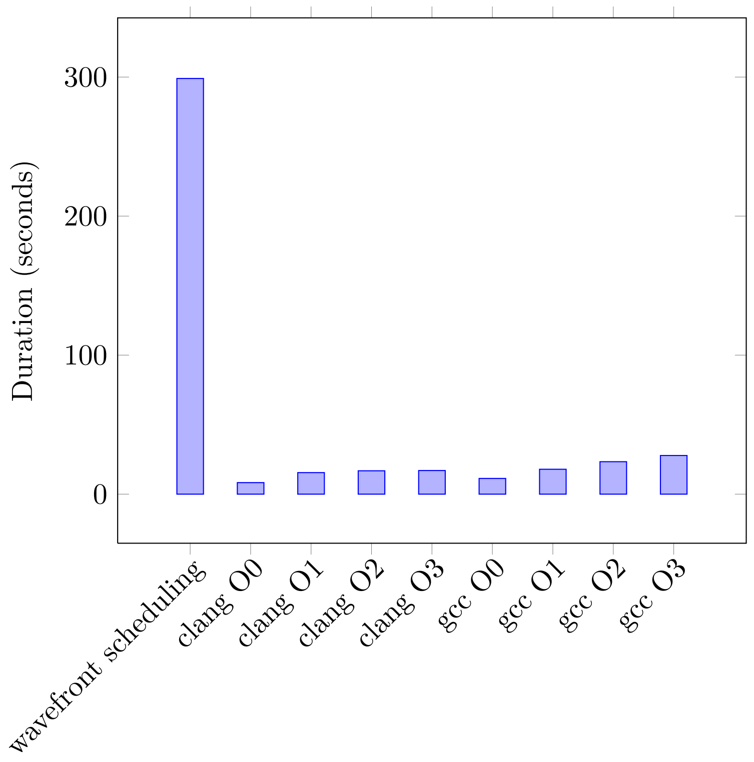

Figure 4.1: Running time of applying wavefront scheduling to CLIPS compared to compiling it with clang and gcc at different optimization levels.

is not meant as a general purpose optimization but one of those optimizations that are applied once at the end of development before release to squeeze as much performance out of the program as possible. With that being said, does the overhead of wavefront scheduling equate to a higher performance program?

Answering this question requires the use of the same basic block scheduler used in verifying that wavefront scheduling is correct. The inefficiency of the basic block scheduler makes it perfect for testing the quality of the code generated by the wavefront scheduler. This basic block scheduler operates in a similar fashion to the basic block scheduler found in the wavefront scheduling implementation. The biggest difference being that it sends a large number of messages and as such becomes quite slow as the number of instructions in the target basic block increases. It also operates on Itanium assembler instead of LLVM bitcode.

This makes scheduling a little more involved due to the fact that register usage throughout the program must be taken into account as well to prevent the generation of invalid code. This program makes a perfect test to determine what kind of a performance change wavefront scheduling brings to the table.

The inputs used were blocks that contained 1000, 1250, 1500, 1750, and 2000 instructions respectively. As the results will show, going higher than 2000 instructions takes a considerable amount of time to complete. As the number of instructions increases the gaps between the different versions of the CLIPS program become more pronounced.

Compiler	Optimization Level	Duration
clang	O0	1m 12.37 seconds
gcc	O0	1m 11.67 seconds
opt with wavefront	O0	1m 0.37 seconds
opt without wavefront	O0	1m 0.14 seconds
opt with wavefront	LTO	45.71 seconds
gcc	O1	40.82 seconds
clang	O1	38.14 seconds
gcc	O2	36.48 seconds
opt without wavefront	LTO	36.39 seconds
clang	O3	35.31 seconds
clang	O2	34.78 seconds
gcc	O3	34.39 seconds
opt with wavefront	O1	34 seconds
opt without wavefront	O2	33.96 seconds
opt without wavefront	O1	33.82 seconds
opt with wavefront	O2	33.82 seconds
opt with wavefront	O3	33.77 seconds
opt without wavefront	O3	33.60 seconds

Table 4.2: Results of basic block scheduling for 1000 instructions sorted by duration

Table 4.2 represents the running time of the basic block scheduler for 1000 instructions. Wavefront scheduling at O0 shaves 12 and 11 seconds off of the running times of clang O0 and gcc O0 respectively. While not nearly as significant, it is slower than opt without wavefront scheduling with O0 by 0.2 seconds. This reduction in running time is most likely due to the fact that the linker may have access to alignment information missing from clang and gcc at O0.

Wavefront scheduling with LTO outperformed opt without wavefront scheduling with O0 by 14.43 seconds but lost to gcc 01, clang 01, gcc O2, and opt without wavefront scheduling with LTO by 4.89, 7.57, 9.23, and 9.32 seconds respectively. This reduction in speed is most likely due to the fact that the application of wavefront scheduling is done under ideal conditions with an infinite number of registers. On an architecture like x86-64 that only has 16 general purpose registers, this can translate to the need to access memory more often.

Wavefront scheduling with O1 outperforms clang O3, clang O2, and gcc O3 by 1.31, 0.78, and 0.39 seconds respectively. It is slower than the version of opt without wavefront scheduling with O1 by 0.18 seconds. This loss in performance is most likely due to the limited number of registers in the x86-64 architecture.

Wavefront scheduling with O2 actually outperforms opt without wavefront scheduling with O2 by 0.14 seconds. Interestingly enough the O1 version of opt without wavefront scheduling ran at the exact same speed at as the wavefront scheduled O2 version. It would seem that the application of O2 combined with wavefront scheduling is a good combination.

Wavefront Scheduling with O3 had the second fastest running time for 1000 instructions. It was outperformed by opt without wavefront scheduling at O3 by 0.17 seconds. It would be safe to say that O3 compensates for the reduction in performance generated by wavefront scheduling.

At 1000 instructions, wavefront scheduling shows that it does reduce performance by itself and less so when combined with other optimizations. While opt without wavefront scheduling is faster, the difference in speed at O2 and O3 are slight. The performance decrease is most likely due to the fact that applying wavefront scheduling will convert a basic block into a series of interwoven chains. Each link in the chain represents an instruction and having several links next to each other exposes parallelism to the processor during execution. While a setup like this is ideal for a processor with a large number of registers, it is less than ideal for a register starved architecture like x86-64.

Table 4.3 on the next page represents the running time of the basic block scheduler for 1250 instructions. Wavefront scheduling with O0 outperforms clang O0, and gcc O0 by wavefront scheduling O0 by 23.83 and 23.75 respectively. It looses to opt without wavefront scheduling with O0 by 0.34 seconds. This behavior is similar to what was seen at 1000 instructions.

Compiler	Optimization Level	Duration
clang	O0	2 minutes 23.92 seconds
gcc	O0	2 minutes 23.84 seconds
opt with wavefront	O0	2 minutes 00.09 seconds
opt without wavefront	O0	1 minutes 59.75 seconds
opt with wavefront	LTO	1 minutes 35.50 seconds
gcc	O2	1 minutes 33.32 seconds
gcc	O1	1 minutes 16.42 seconds
clang	O1	1 minutes 15.43 seconds
opt without wavefront	LTO	1 minutes 11.25 seconds
gcc	O3	1 minutes 09.10 seconds
clang	O3	1 minutes 09.01 seconds
clang	O2	1 minutes 08.90 seconds
opt without wavefront	O2	1 minutes 07.70 seconds
opt with wavefront	O1	1 minutes 07.54 seconds
opt with wavefront	O3	1 minutes 07.29 seconds
opt without wavefront	O1	1 minutes 07.25 seconds
opt with wavefront	O2	1 minutes 07.14 seconds
opt without wavefront	O3	1 minutes 06.85 seconds

Table 4.3: Results of basic block scheduling for 1250 instructions sorted by duration

Wavefront scheduling with LTO is in the same position relative to what was seen in Table 4.2 on page 180. It is outperformed by gcc O2, gcc O1, clang O1, and opt without wavefront scheduling with LTO by 2.18,19.08, 20.07, and 24.25 seconds respectively. However, it does outperform opt without wavefront scheduling with O0 by 24.07 seconds. It is suprising to see gcc O2 being out performed by gcc O1 and clang O1 by such a large margin. This is a partial reverse of the ordering seen in Table 4.2 on page 180. There are two possible reasons for this shift in behavior:

1. A fluke
2. The optimizations gcc applies at O2 may not scale well.

It is probably the first case but further research would be needed to figure out why this reversal in running time occured.

Wavefront scheduling with O1 outperforms gcc O3, clang O3, clang O2, and opt without wavefront scheduling with O2 by 1.56, 1.47, 1.36, and 0.16 seconds respectively. The performance improvement over opt without wavefront scheduling with O2 is small but suprising as it shows that the application of O2 may not always yield a performance benefit.

Wavefront scheduling with O3 is not the fastest version this time. It outperforms wavefront scheduling with O1 by 0.25 seconds. This tiny reduction in running time is pretty important when running an NP-Complete operation such as basic block scheduling. Each decrease in overall running time can translate to rather large improvements with larger data sets.

Wavefront scheduling with O2 outperforms opt without wavefront scheduling with O1 by 0.11 seconds. However, it is slower than opt without wavefront scheduling with O3 by 0.29 seconds. This is most likely due to the fact that O3 contains optimizations targeting loops is most likely the reason for the difference speed.

This test shows that wavefront scheduling does provide improvements over gcc and clang but is still slower than opt without wavefront with O3. Once again, wavefront scheduling with O2 stood out as being extremely fast.

Compiler	Optimization Level	Duration
gcc	O0	4 minutes 18.43 seconds
clang	O0	4 minutes 17.93 seconds
opt without wavefront	O0	3 minutes 36.31 seconds
opt with wavefront	O0	3 minutes 36.31 seconds
opt with wavefront	LTO	3 minutes 00.84 seconds
gcc	O1	2 minutes 17.96 seconds
clang	O1	2 minutes 13.15 seconds
gcc	O2	2 minutes 12.53 seconds
opt without wavefront	LTO	2 minutes 08.21 seconds
gcc	O3	2 minutes 05.03 seconds
clang	O3	2 minutes 04.14 seconds
clang	O2	2 minutes 03.17 seconds
opt without wavefront	O2	2 minutes 01.79 seconds
opt with wavefront	O1	2 minutes 01.68 seconds
opt with wavefront	O3	2 minutes 00.96 seconds
opt without wavefront	O3	2 minutes 00.92 seconds
opt with wavefront	O2	2 minutes 00.84 seconds
opt without wavefront	O1	2 minutes 00.75 seconds

Table 4.4: Results of basic block scheduling for 1500 instructions sorted by duration

Table 4.4 represents the running time of the basic block scheduler for 1500 instructions. Wavefront scheduling with O0, once again, out performed gcc O0 and clang O0 by 42.12

and 41.62 seconds respectively. Unlike the previous two tables, the running time of opt without wavefront with O0 is the same as wavefront scheduling with O0. This is either a fluke or the NP-Complete aspect of basic block scheduling is making any performance loss by wavefront scheduling null and void.

Like the previous two tables, the application of wavefront scheduling with LTO is slower than , gcc O1, clang O1, gcc O2, and opt without wavefront scheduling with LTO by 42.88, 47.69, 48.31, and 52.13 seconds respectively. Wavefront scheduling with LTO was faster than opt without wavefront scheduling with O0 by 35.47 seconds. It is important to note that gcc O2 is once again faster than gcc O1 and clang O1. This shows that wavefront scheduling with LTO is consistently slower than gcc O1, clang O1, gcc O2, and opt without wavefront scheduling with LTO.

Wavefront scheduling with O1 is faster than gcc O3, clang O3, clang O2, and opt without wavefront scheduling with O2 by 3.35, 2.45, 1.49, 0.11 seconds respectively. The performance of this version is consistent with what was seen in Figure 4.3 on page 182.

Wavefront scheduling with O3 outperforms wavefront scheduling with O1 by 0.72 seconds. This is a somewhat significant gap in running time. However, this version is still slower than opt without wavefront scheduling with O3 by 0.04 seconds. While this difference may seem small, it means that the application of scheduling is slowing down CLIPS slightly.

Once again, wavefront scheduling with O2 is the second fastest version of CLIPS that is faster than opt without wavefront scheduling with O3 by 0.08 seconds. This shows that the application of O3 with wavefront scheduling most likely *slows* the program down. However, it is still slower than opt without wavefront scheduling with O1 by 0.09 seconds. The application of O2, which contains optimizations that usually improve performance, seems to be the best partner to wavefront scheduling for CLIPS.

Overall, this test is putting creedence to the idea that wavefront scheduling can improve performance when coupled with O2 but it will never be the fastest.

Table 4.5 on the following page represents the running time of the basic block scheduler for 1750 instructions. Once again, wavefront scheduling with O0 outperforms gcc O0 and clang O0 by a large marging while losing to opt without wavefront scheduling with O0 by

Compiler	Optimization Level	Duration
gcc	O0	7 minutes 08.65 seconds
clang	O0	7 minutes 07.88 seconds
opt with wavefront	O0	5 minutes 58.50 seconds
opt without wavefront	O0	5 minutes 58.29 seconds
opt with wavefront	LTO	5 minutes 06.50 seconds
gcc	O1	3 minutes 49.12 seconds
gcc	O2	3 minutes 42.12 seconds
clang	O1	3 minutes 41.45 seconds
opt without wavefront	LTO	3 minutes 32.89 seconds
gcc	O3	3 minutes 29.87 seconds
clang	O3	3 minutes 25.34 seconds
clang	O2	3 minutes 24.28 seconds
opt without wavefront	O2	3 minutes 22.68 seconds
opt with wavefront	O3	3 minutes 21.43 seconds
opt with wavefront	O1	3 minutes 21.28 seconds
opt with wavefront	O2	3 minutes 21.14 seconds
opt without wavefront	O1	3 minutes 20.61 seconds
opt without wavefront	O3	3 minutes 19.77 seconds

Table 4.5: Results of basic block scheduling for 1750 instructions sorted by duration

a small margin. This behavior is quite consistent with what has been seen in the previous three tables.

Wavefront scheduling with LTO is outperformed gcc O1, gcc O2, clang O1, and opt without wavefront scheduling with LTO by 77.38, 84.38, 85.05, and 93.61 seconds respectively. It is becoming increasingly obvious that wavefront scheduling with link time optimization will slow things down.

Unlike the last test, wavefront scheduling with O3 is slower than the corresponding O1 and O2 versions; However, it is faster than gcc O3, clang O3, clang O2, and opt without wavefront scheduling with O2 by 8.44, 3.91, 2.85, and 1.25 seconds respectively. This shows that the application of O3 does not always yield performance improvements in all cases for a given program.

Wavefront scheduling with O1 outperforms wavefront scheduling with O3 by 0.15 seconds. It is slower than wavefront scheduling with O2 and the O1 and O3 versions of opt without wavefront scheduling but that is to be expected from the trends of the previous three tables.

Wavefront scheduling with O2 is once again the fastest wavefront scheduled version. However, it is bested by the O1 and O3 versions of opt with wavefront scheduling by 0.52 and 1.36 seconds respectively. This would lead to the conclusion that the scheduling being performed is hampering performance in one way or another. Further research is required to figure out exactly what is going on.

Once again, the O1 and O3 versions of wavefront scheduling are slower than the O2 version. This makes maximum compatibility with the O2 set of optimizations to be a top priority for the next version of the wavefront scheduler.

Compiler	Optimization Level	Duration
gcc	O0	11 minutes 18.72 seconds
clang	O0	11 minutes 14.50 seconds
opt with wavefront	O0	9 minutes 26.73 seconds
opt without wavefront	O0	9 minutes 25.96 seconds
opt with wavefront	LTO	8 minutes 19.62 seconds
gcc	O1	6 minutes 02.37 seconds
gcc	O2	5 minutes 51.75 seconds
clang	O1	5 minutes 49.11 seconds
opt without wavefront	LTO	5 minutes 33.62 seconds
gcc	O3	5 minutes 33.54 seconds
clang	O3	5 minutes 27.13 seconds
clang	O2	5 minutes 20.59 seconds
opt with wavefront	O3	5 minutes 19.09 seconds
opt with wavefront	O2	5 minutes 18.78 seconds
opt without wavefront	O2	5 minutes 18.50 seconds
opt with wavefront	O1	5 minutes 18.05 seconds
opt without wavefront	O1	5 minutes 18.00 seconds
opt without wavefront	O3	5 minutes 17.41 seconds

Table 4.6: Results of basic block scheduling for 2000 instructions sorted by duration

Table 4.6 represents the running time of the basic block scheduler for 2000 instructions. Like the previous four tables, the application of wavefront scheduling with O0 yields a program that always runs faster than gcc O0 and clang O0 while being slower than opt without wavefront scheduling with O0.

Wavefront scheduling with LTO, once again, is slower than gcc O1, gcc O2, clang O1, and opt without wavefront scheduling with LTO by 137.25, 147.87, 150.51, and 166 seconds respectively.

Interestingly enough, wavefront scheduling with O3 is slower than the O1 and O2 versions of wavefront scheduling. However, it is faster than gcc O3, clang O3, and clang O2 by 14.45, 8.04, and 1.5 seconds respectively. It would seem that as the number of instructions increases the optimizations applied by O3 cause a drop in performance when coupled with wavefront scheduling.

Unlike the previous four tests, this test has wavefront scheduling with O2 being slower than wavefront scheduling with O1 and opt without wavefront scheduling with O2 by 0.73 and 0.28 seconds respectively. This would tend to show that the optimizations applied by O2 begin to *slow* CLIPS down. Further testing is required to see if the performance loss is constant as the number of instructions rises.

The application of wavefront scheduling with O1 is the fastest version in this table. However, it is still bested by the O1 and O3 versions of opt without wavefront scheduling by 0.05 and 0.64 seconds respectively. This means that wavefront scheduling does degrade performance comparatively with the opt version of the code.

The application of wavefront scheduling is a mixed bag with respect to performance improvements in CLIPS. While it consistently outperforms clang and gcc when coupled with O1, O2, or O3 it does a miserable job if combined with LTO. It also seems that wavefront scheduling also *breaks* the performance stability of O2 and O3 as the number of instructions rise. This requires further research by testing at 2250 through 3000 instructions.

Figure 4.2 on the following page, shows the average running time for each compiler across the optimizations O0 through O3. Unlike the individual tests, the combined version shows that wavefront scheduling is quite close in performance to opt without wavefront scheduling. This is because the difference is speed is usually quite minute between the two compilers. On average, wavefront scheduling is slower than opt without wavefront scheduling.

The testing results with CLIPS shows that wavefront scheduling combined with other optimizations can improve performance over clang and gcc. However, when compared to opt without wavefront scheduling it is either slower or on par. This can be attributed not only to the lack of registers in x86-64 but also the fact that applying basic block scheduling, as it is done in this implementation, will require more loads and stores to occur to keep track of the intermediate results that are being computed in a piecemeal fashion. However, even

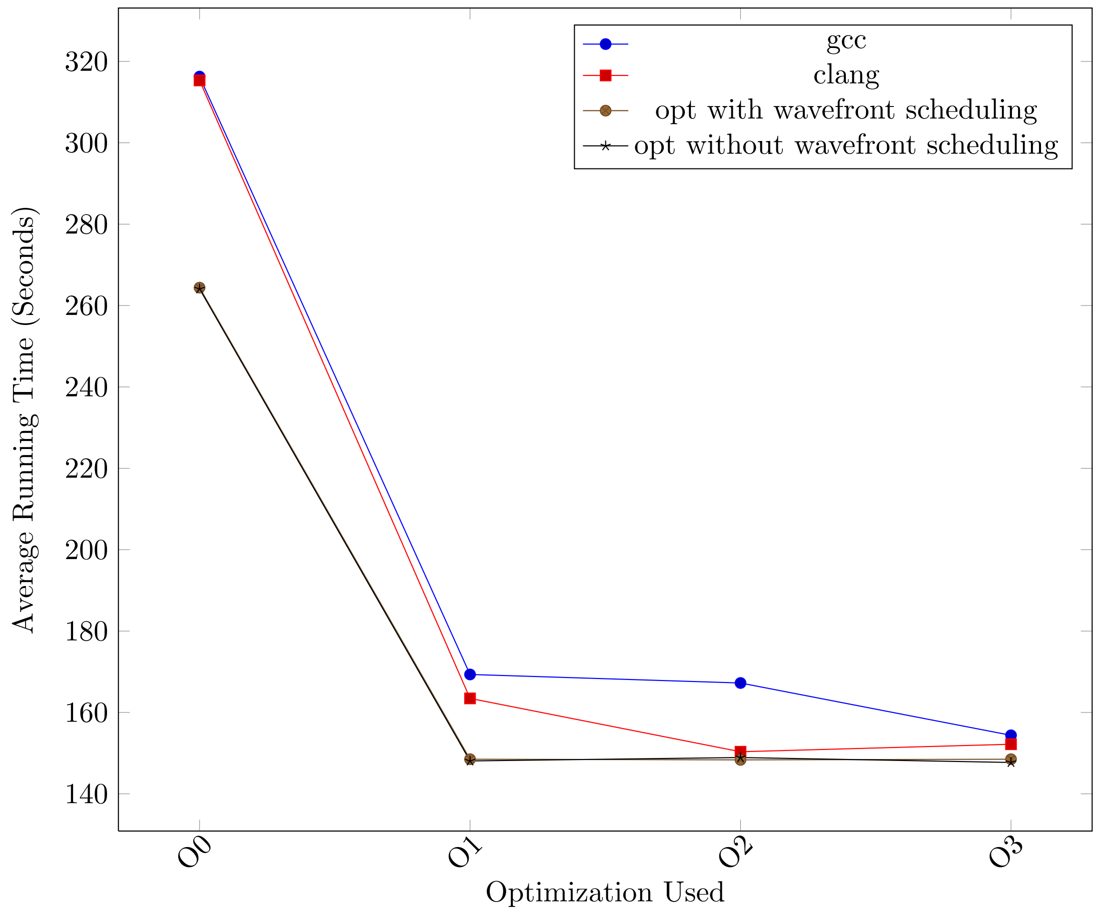

Figure 4.2: Average running time of CLIPS for each compiler for O0 through O3

with that being said, the amount of overhead associated with applying wavefront scheduling to CLIPS yielded an appreciable performance benefit whenc combined with O2.

4.3.2 FLOPS

FLOPS is a floating point analysis tool designed to provide a cross architecture way of determining the floating point capabilities of the given processor. It is divided up into eight modules that test the floating point unit (FPU) of the processor in different ways[8]. The result of each module is given in megaFLOPS which denotes how well the given processor handled it. While FLOPS is not a large program, it is an effective test of how well the wavefront scheduler handles floating point instructions. Figure 4.3 shows the amount of time taken to schedule FLOPS compared to compilation time by clang and gcc at different optimization levels.

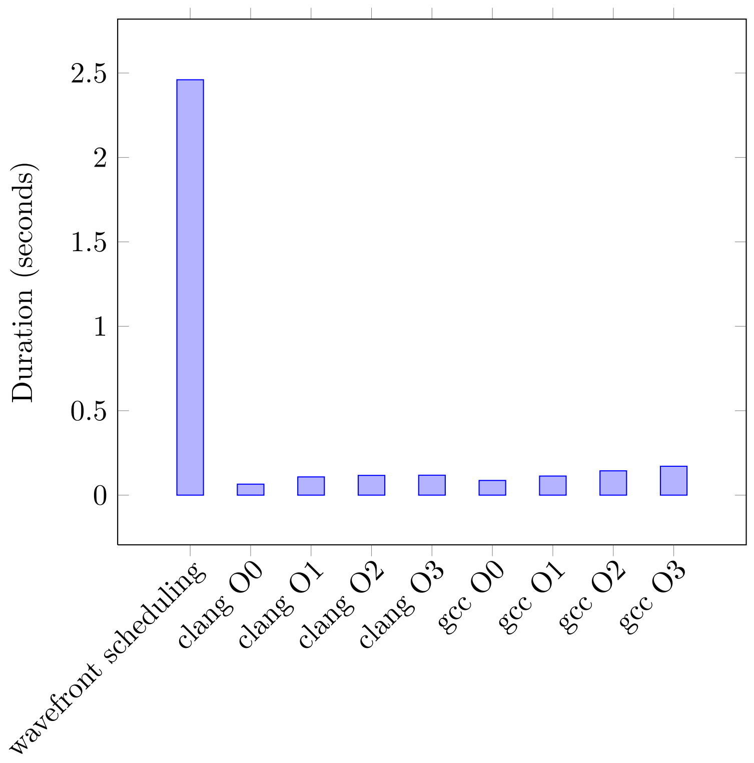

Figure 4.3: Running time of applying wavefront scheduling to FLOPS compared to compiling it with clang and gcc at different optimization levels.

It shows an almost exact likeness to the running time comparison for CLIPS. While the duration is far shorter the difference is quite noticeable. Even for the simplest of programs, wavefront scheduling will add a significant amount of overhead. It may be believed that the overhead associated with the KCE would be the culprit but that is not the case. Usually the KCE only adds about 0.1 seconds to a program like this. The real culprit is the use of CLIPS and the practical *abuse* of multifields and the continual modification of state.

But does the overhead equate to an improvement in performance?

This question can only be answered by comparing the different MFLOPS values computed from each variant of the test across the different compiled versions. This program is also interesting in that it is up to the compiler to identify sections of code that can be *vectorized* which allows different portions to be executed in parallel. The biggest killer to floating point performance is the use of floating point divides as they are expensive in terms of execution resources and time. As stated earlier there are eight modules.

The first module of FLOPS calculates the integral of a provided function [1] f, that has a result of $ln(f(1))$. The operation contains seven double precision floating point adds, zero subtracts, six multiplies, and one divide[8]. This module is meant to test how well the processor does with code that is add and multiply heavy with a touch of divide thrown in. The results of this module are shown in Figure 4.4.

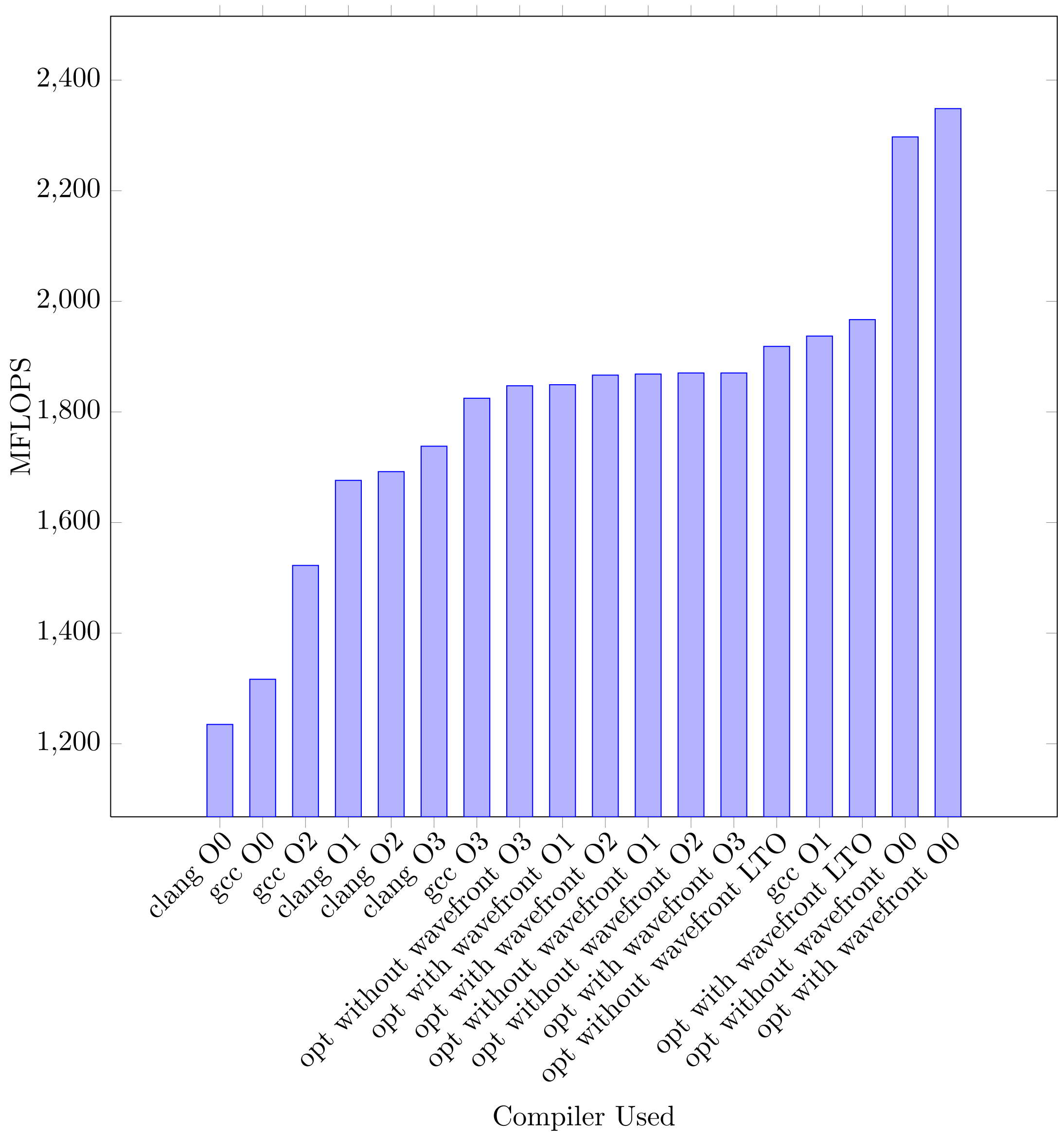

Figure 4.4: Bar graph representing determined MFLOPS for module 1. The values are organized from lowest MFLOPS to highest.

The results of module 1 were somewhat surprising. Wavefront scheduling with O0

[1]The actual function could not be easily determined from the source code.

yielded the highest MFLOPS value with opt without wavefront scheduling with O0 following. It is interesting to see that applying optimizations such as O1, O2, O3, and LTO actually resulted in a performance drop. The reason for this can be attributed to the fact that the different compilers may be over-optimizing the code.

With the exception of gcc O1, the application of wavefront scheduling yielded higher quality code compared to gcc and clang at the same optimization levels. This gives more credit to the idea that wavefront scheduling is a viable optimization for superscalar processors.

The second module calculates the value of π from the Taylor Series expansion of $atan(1.0)$. There are seven double precision operations per iteration which consist of three additions, two subtractions, one multiply, and one divide[8]. The results of this module are demonstrated by Figure 4.5 on the following page.

Unlike the first module, this module shows that wavefront scheduling with O0 actually reduced performance. In fact, coupling wavefront scheduling with any other optimization caused the performance to drop greatly. The most likely reason for the change in performance has to do with the fact that this module is designed to be very difficult to vectorize due to the code being very sequential in nature. The application of wavefront scheduling does very little to resolve this issue and seems to make it worse. Clang O0 and gcc O0 are the fastest because they do not make changes to the code. This allows the processor to completely handle identifying parallelism in this module which it seems to do a better than any optimization. It is interesting to note that gcc O2 and opt without wavefront scheduling with LTO do a miserable job of optimizing this module. This module shows that sometimes the application of any optimization can cause a reduction in performance for a given CPU.

The third module calculates the integral of $sin(x)$ from 0.0 to $\frac{\pi}{3.0}$ using the Trapezoidal Method. The result of this integration will be 0.5. Computing this value requires 17 operations per iteration consisting of six additions, two subtractions, nine multiplications, and zero divisions[8]. This is a very multiplication heavy module designed to determine how efficient the processor's floating point unit is at multiplication. The results of this module are shown in Figure 4.6 on page 193.

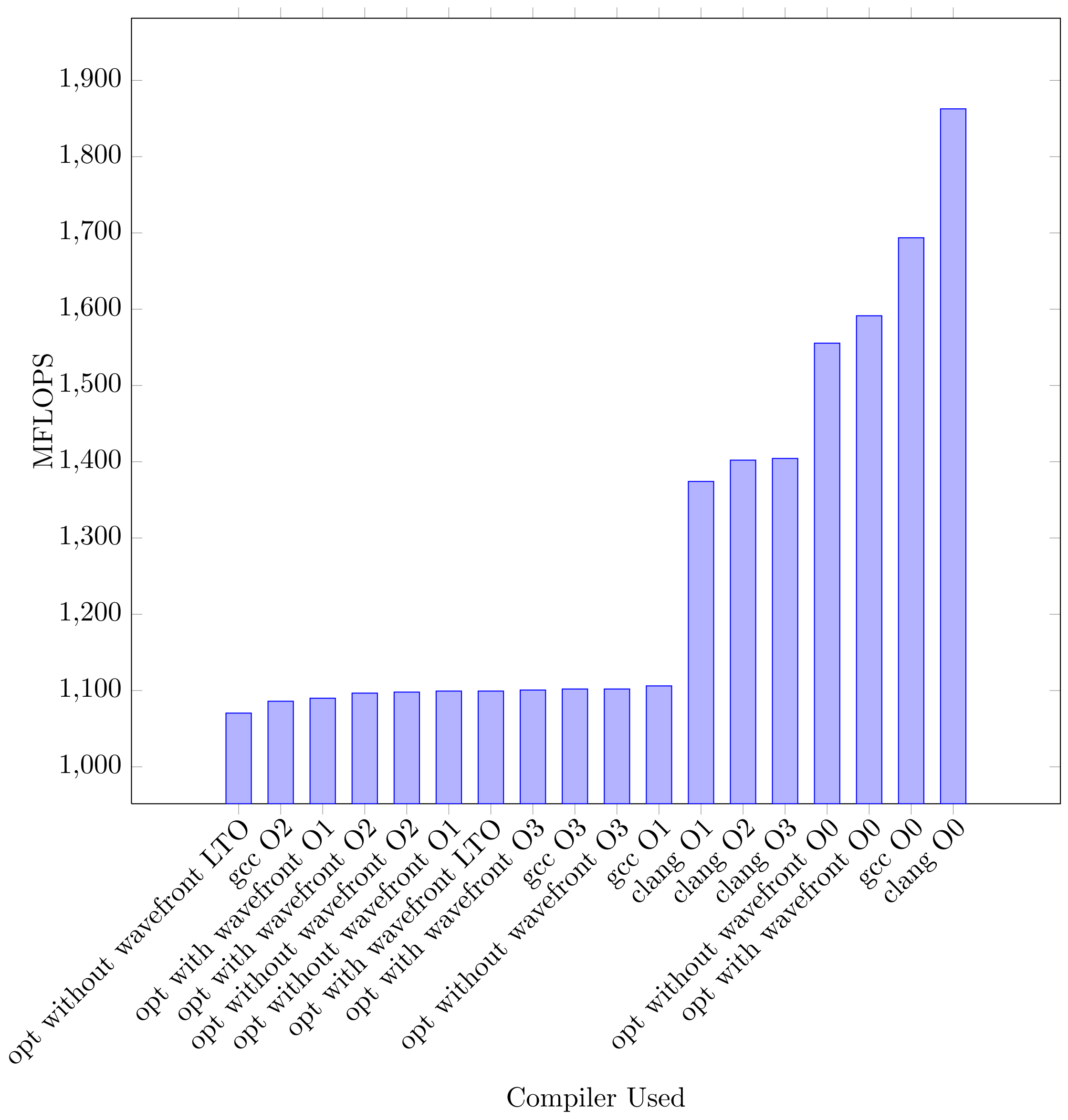

Figure 4.5: Bar graph representing determined MFLOPS for module 2

The application of wavefront scheduling improves the performance of this module compared to gcc and clang. The only outlier in this case is gcc O1 which seems to do a fantastic job of using the processors FPU to its fullest. This module also shows that wavefront scheduling with O0 is faster than the majority of the other versions. This is most likely due to the fact that instructions are scheduled into *chains* by the wavefront scheduler. Take the original version of this fragment of code from [8] described in Figure 4.7 on page 194.

Assuming that this code makes up a single block, it is possible to apply basic block

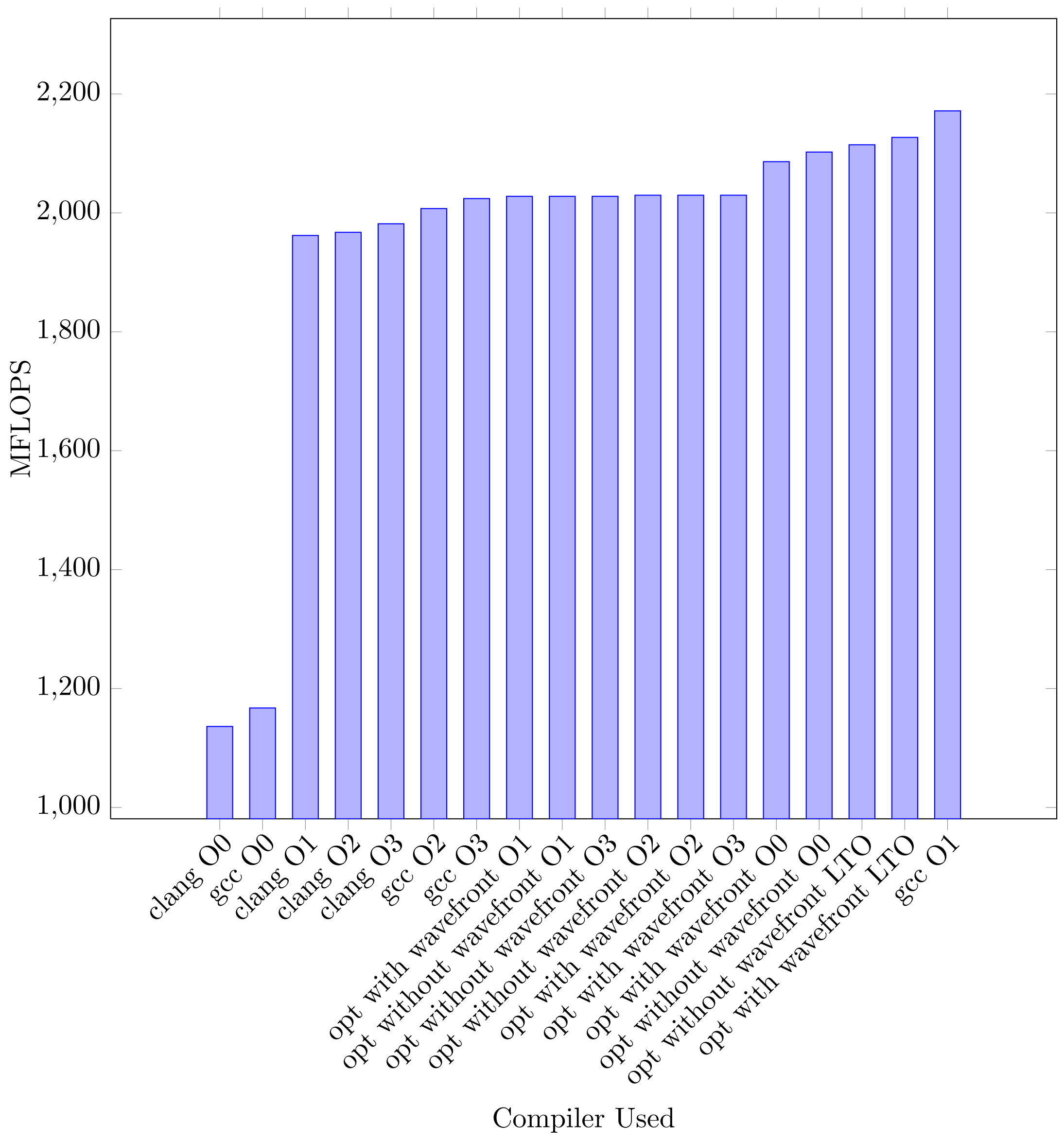

Figure 4.6: Bar graph representing determined MFLOPS for module 3

scheduling to this basic block to great effect. This is demonstrated by Figure 4.8 on the next page.

While there are portions of this code that could be further scheduled, the point of the matter is that the code itself has a lot of scheduling potential. This potential is what allows wavefront scheduling to shine. The fourth module calculates the integral of $cos(x)$ from 0.0 to $\frac{\pi}{3}$ using the Trapezoidal Method with the result being $sin(\frac{\pi}{3})$. There are 15 operations per iteration consisting of seven additions, zero subtractions, eight multiplications, and zero

```
T[9]   = T[1] * TimeArray[1] - nulltime;

u   = piref / three;
w   = u * u;
sa  = u * ((((((A6*w-A5)*w+A4)*w-A3)*w+A2)*w+A1)*w+one);

T[10] = T[9] / 17.0;                              /********************/
sa = x * ( sa + two * s ) / two;                  /* sin(x) Results.  */
sb = 0.5;                                         /********************/
sc = sa - sb;
T[11] = one / T[10];
```

Figure 4.7: The original version of a fragment of code from flops.c

```
T[9]   = T[1] * TimeArray[1] - nulltime;
sb = 0.5;                                         /********************/
T[10] = T[9] / 17.0;                              /********************/
u   = piref / three;
T[11] = one / T[10];
w   = u * u;
sa  = u * ((((((A6*w-A5)*w+A4)*w-A3)*w+A2)*w+A1)*w+one);
sa = x * ( sa + two * s ) / two;                  /* sin(x) Results.  */
sc = sa - sb;
```

Figure 4.8: The scheduled version of the fragment of code from flops.c. This version was acquired by simply rearranging statements.

divides. This test is meant to see how well the FPU does with floating point multiply and add operations. The results of this module are described by Figure 4.9 on the following page.

Figure 4.9 on the next page demonstrates that the version of FLOPS that only had wavefront scheduling applied is in the upper echelon of the graph in terms of speed. This is most likely due to the amount of "scheduling material" available. This module also demonstrates that wavefront scheduling coupled with LTO makes for an effective combination. Based on these four modules one could assume that wavefront scheduling may provide a huge performance boost.

The fifth module calculates the integral of $tan(x)$ from 0.0 to $\frac{\pi}{3}$ using the Trapezoidal Method with the result being $ln(cos(\frac{\pi}{3}))$. This module has 29 operations per iteration consisting of 13 additions, zero subtractions, 15 multiplies, and one divide [8]. The results of this module are shown in Figure 4.10 on page 196.

This module shows that wavefront scheduling with O0 does outperform anything gen-

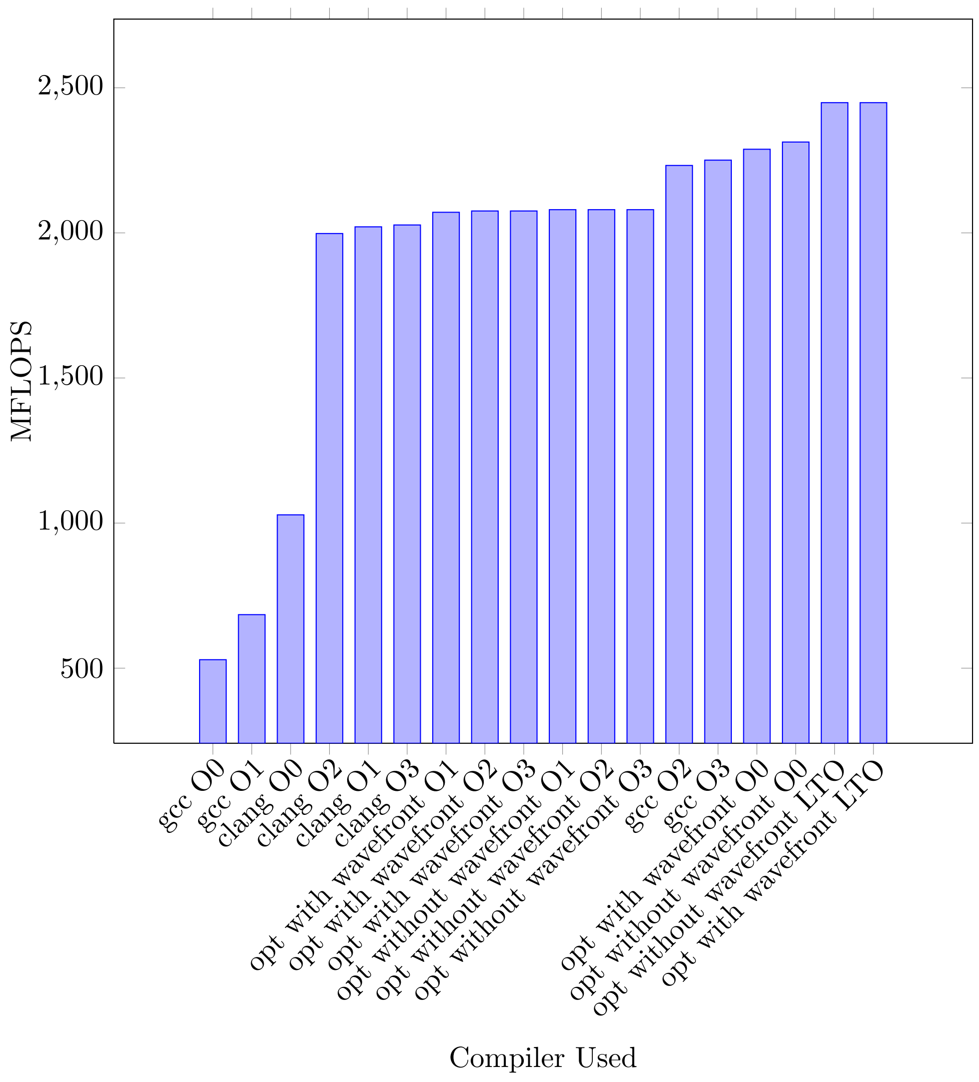

Figure 4.9: Bar graph representing determined MFLOPS for module 4

erated by gcc or clang at any optimization level. However, if it is combined with O1, O2, O3, or LTO then the MFLOPS value increases greatly. This module also shows that the application of wavefront scheduling improves program performance.

The sixth module calculates the integral of $sin(x) * cos(x)$ from 0.0 to $\frac{\pi}{4}$ using the Trapezoidal Method with the result being $sin(\frac{PI}{4})^2$. There are 29 operations per iteration consisting of 13 additions, zero subtractions, 16 multiplications, and zero divides [8]. The results of this module are shown in Figure 4.11 on page 197.

This module shows that the application of LTO combined with wavefront scheduling is the fastest. The application of wavefront scheduling with O0 yields code that is slow but

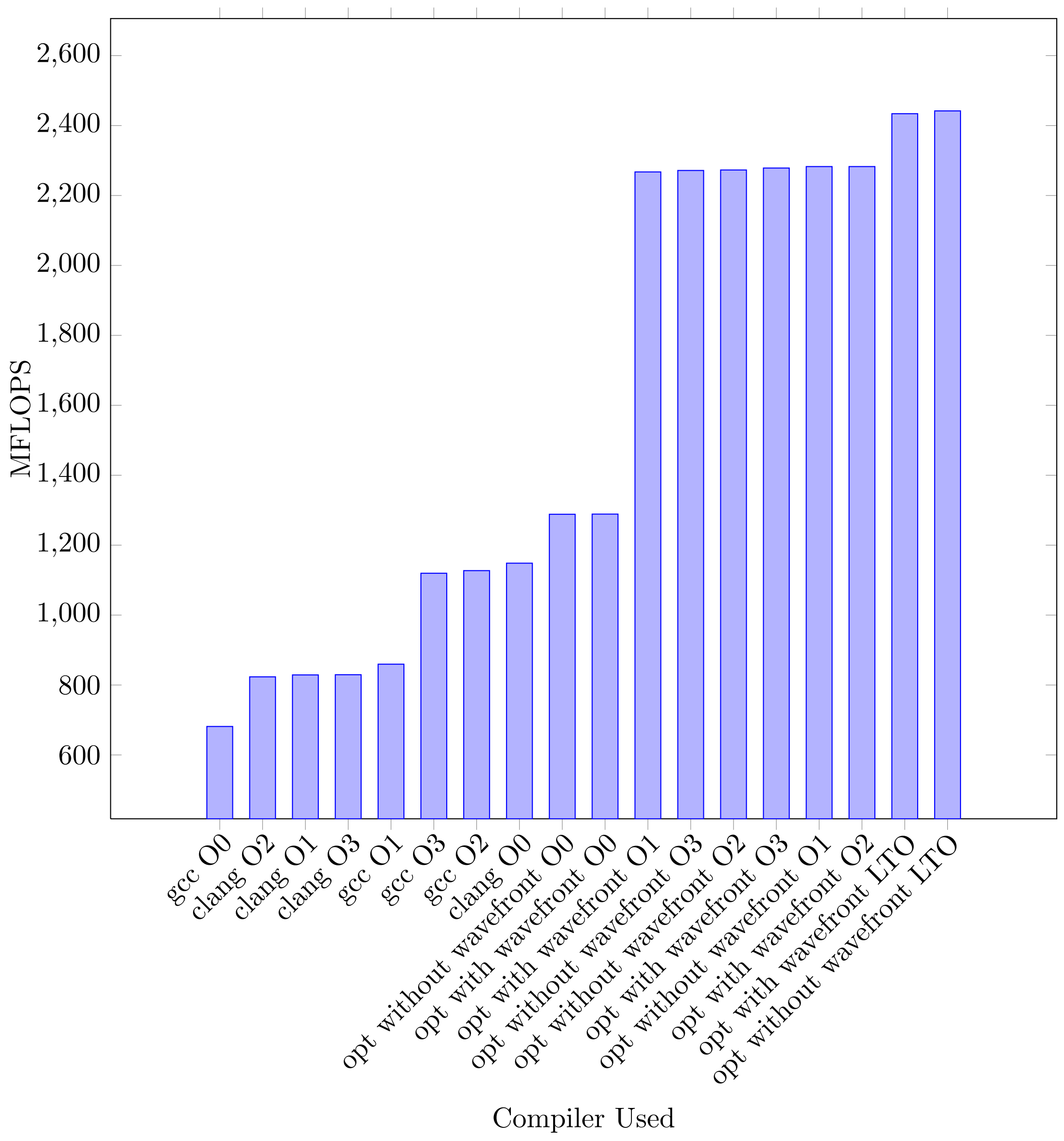

Figure 4.10: Bar graph representing determined MFLOPS for module 5

still out performs clang at O1, O2, and O3. However, with that being said, this module also shows that wavefront scheduling combined with any level of optimization will improve performance greatly.

The seventh module calculates the value of the definite integral from 0 to sa of $\frac{1}{x+1}$, $\frac{x}{x^2+1}$, and $\frac{x^2}{x^3+1}$ using the Trapezoidal Method. There are 12 operations per iteration consisting of three additions, three subtractions, three multiplications, and three divisions [8]. This is the most computationally expensive module due to the use of three divides per iteration.

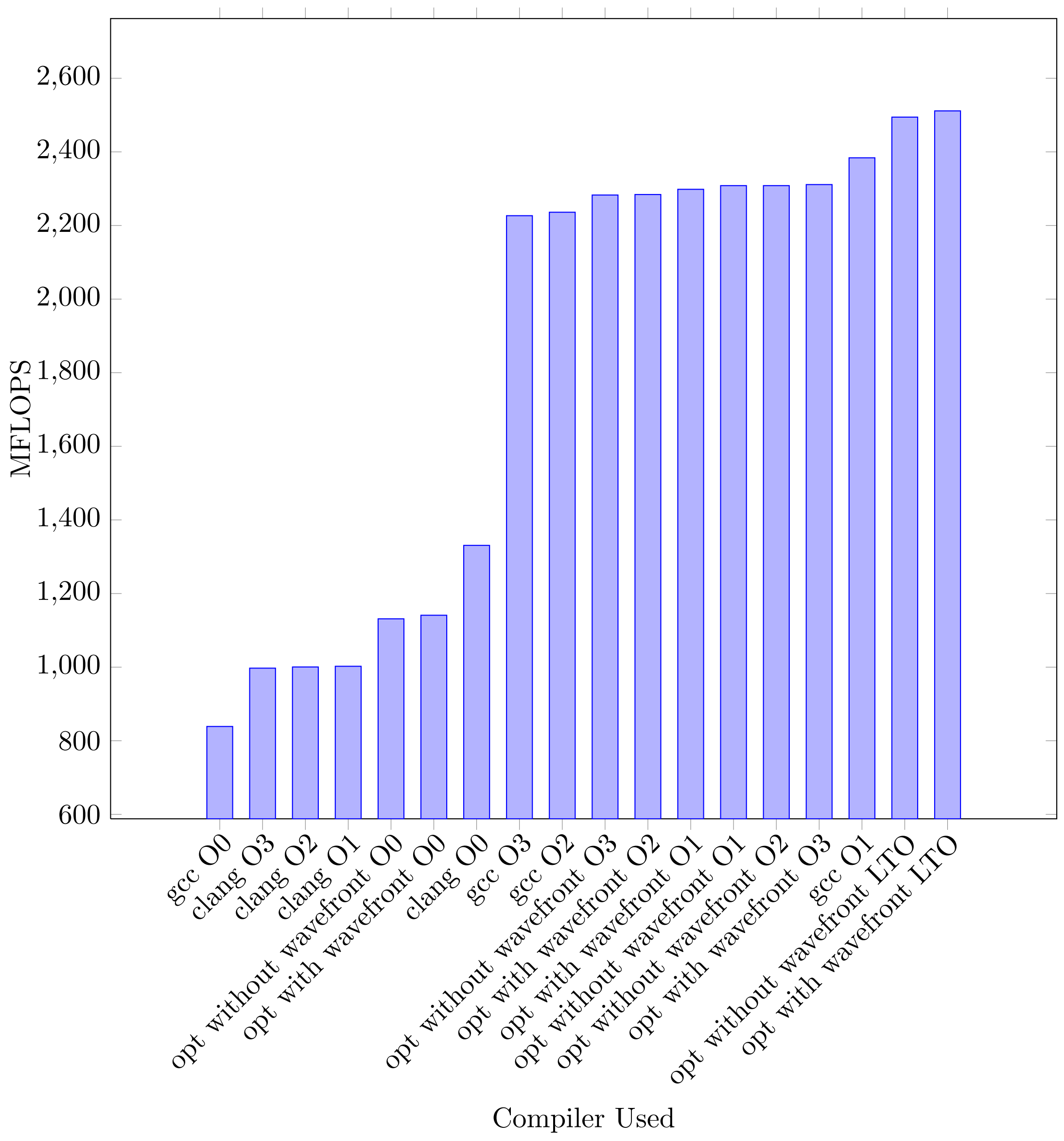

Figure 4.11: Bar graph representing determined MFLOPS for module 6

The results of this module are described in Figure 4.12 on the following page.

This module is designed to be difficult to compute by including a large number of division operations. The application of any optimizations seems to actually cause performance to drop greatly. This is probably due to the fact that the optimizations *reduce* the amount of information available to the code generator which affects what instructions to emit. The only exception to this is gcc which seems to handle divides quite well. However, wavefront scheduling with O0 seems to provide the best performance.

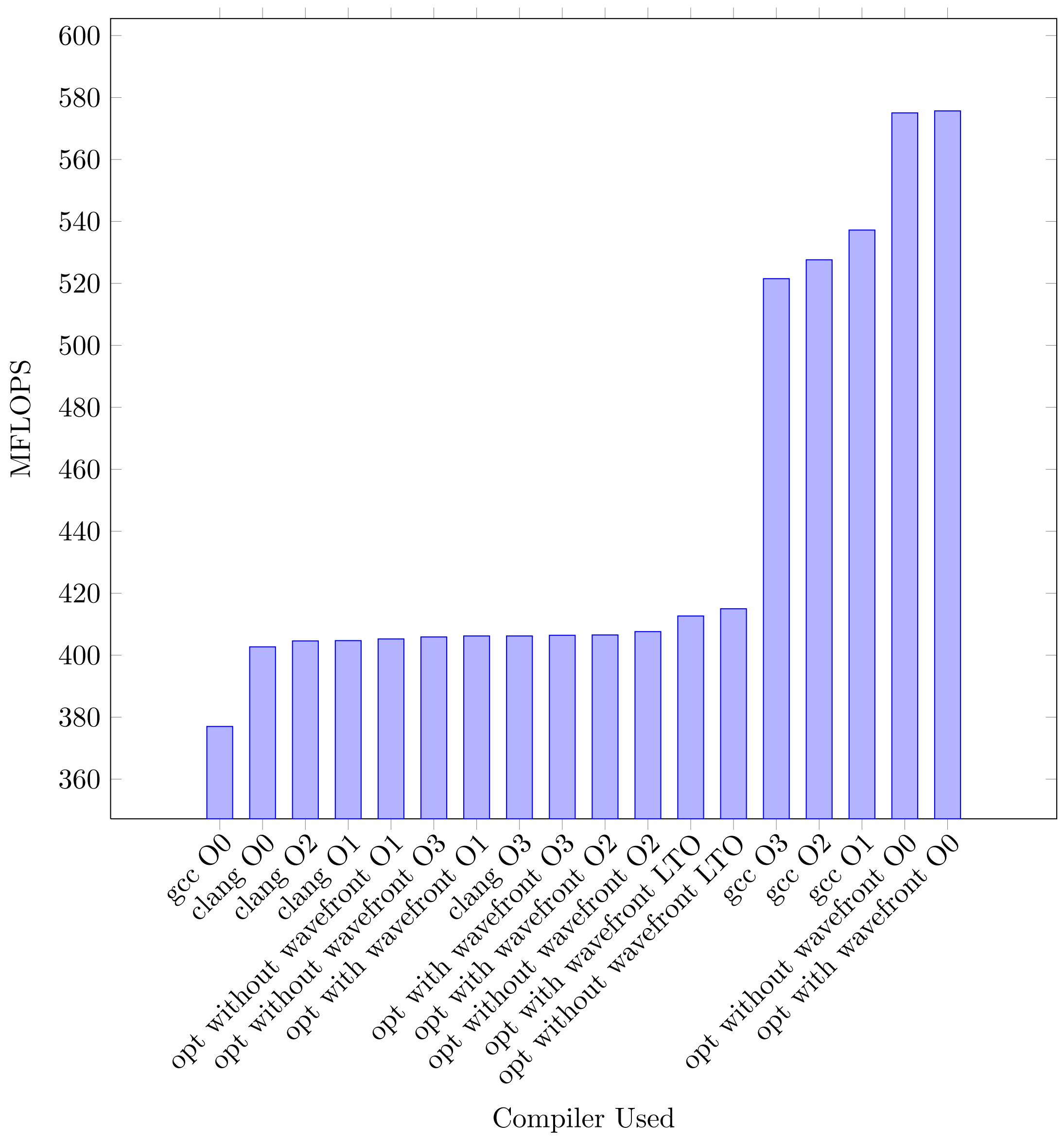

Figure 4.12: Bar graph representing determined MFLOPS for module 7

The eighth and final module calculates the integral of $sin(x) * cos(x) * cos(x)$ from 0 to $\frac{\pi}{3}$ using the trapezoidal method with the result being $\frac{(1-cos(\frac{\pi}{3})^3)}{3}$. There are 30 operations per iteration which consist of 13 additions, zero subtractions, 17 multiplications, and zero divides[8]. Figure 4.13 on the next page shows the results of this module.

This module, like the sixth, shows that wavefront scheduling can only go so far on its own before requiring assistance. However, even with that being said, the wavefront scheduled only version of FLOPS outperformed clang at O3, O1, and O2 by a large margin.

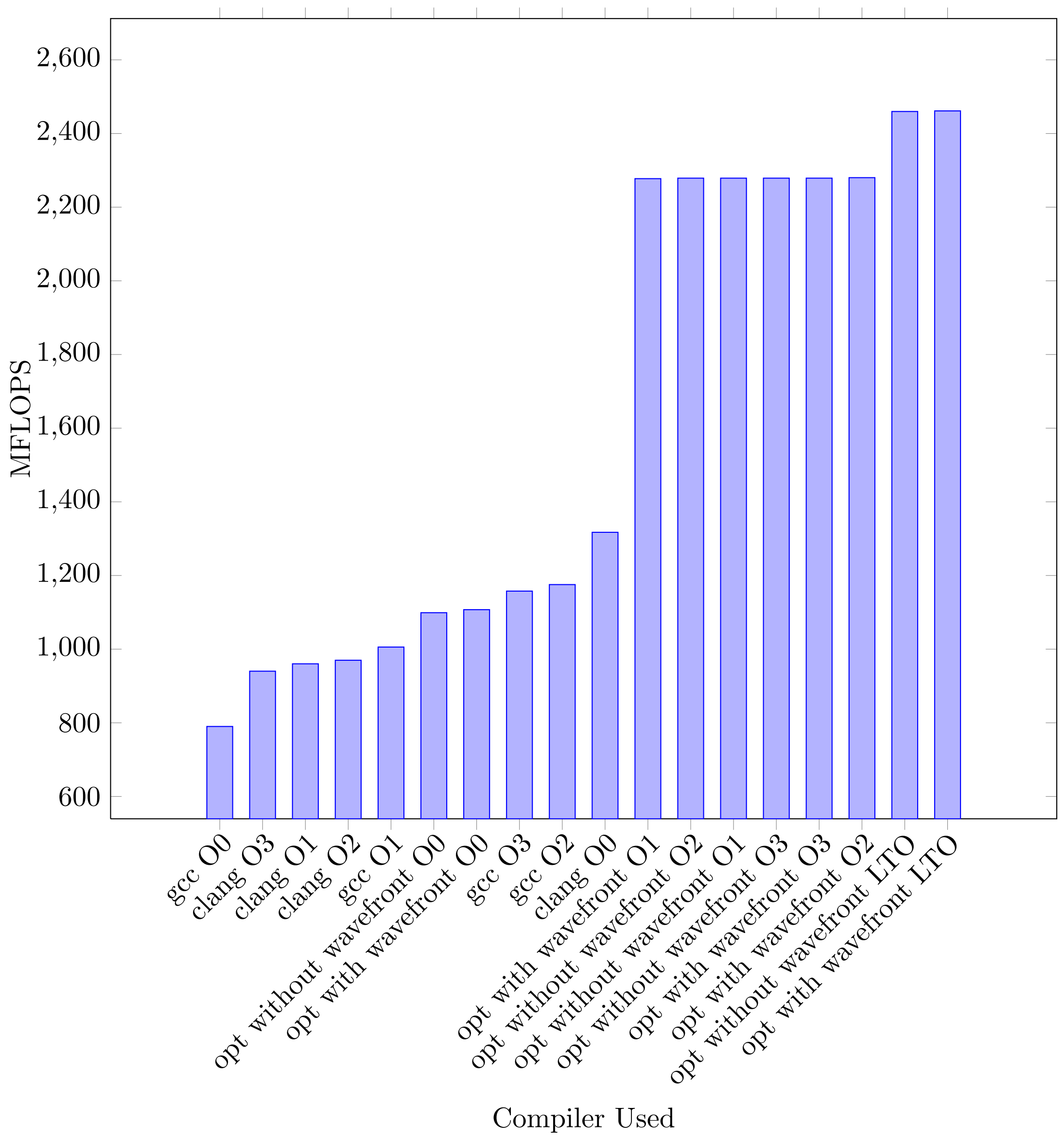

Figure 4.13: Bar graph representing determined MFLOPS for module 8

What is interesting is that there is very little difference in performance between the link time optimized versions of flops with or without wavefront scheduling applied.

Figure 4.14 on the following page, shows the average MFLOPS on each module across the different compilers.

Figure 4.14 shows that on average wavefront scheduling provides very little performance benefit over opt without wavefront scheduling. However, wavefront scheduling usually performs better clang and gcc.

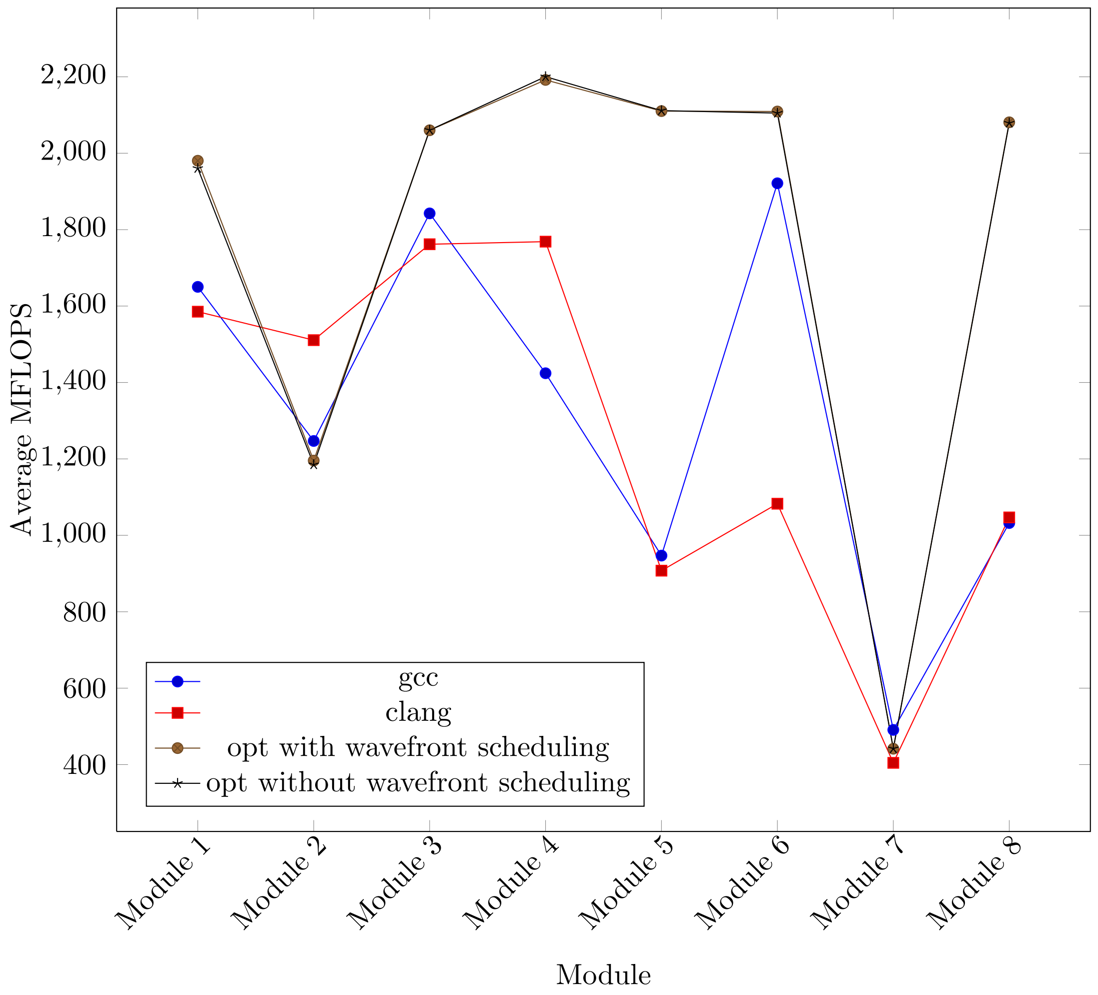

Figure 4.14: Average MFLOPS values for the different modules in FLOPS for each compiler

It is important to realize that scheduling a benchmark such as FLOPS can cause instructions from one module to be computed ahead of time in an earlier one. This will cause those modules to be slower but speed up the module that those values were originally a part of. This can make the results of a benchmark like FLOPS somewhat inaccurate if the scheduler is "smart" enough. While the requirement that all instructions below a call instruction stay there was placed to ensure correctness, it did inadvertently provide the wavefront scheduling implementation with a way to show that scheduling does improve performance while maintaing proper program ordering.

The verdict on applying wavefront scheduling to FLOPS shows that the application does yield performance benefits but it is relative to the specific module as well as the optimizations it is coupled with. In general wavefront scheduling does improve performance compared to clang and gcc.

By these tests it is quite obvious that wavefront scheduling does provide a tangable performance improvment. However, the big question is

Is it worth using?

At this point the answer would be yes because it has been shown to work and provide a meaningful performance improvement. However, this optimization is not useful during program testing and creation. This is the kind of optimization that is applied once at the end right before creating a gold master. The amount of overhead associated with applying wavefront scheduling means that it must be done on a machine that is fast and has a lot of memory. The overhead in terms of scheduling time is a drop in the bucket because it only has to be done once and it means that there is a performance improvement to be had.

Chapter 5

Miscellaneous

5.1 Introduction

This section is devoted to not only implementation specific caveats but future improvements as well.

5.2 Caveats

Introduction

While writing the implementation for this thesis, a number of issues came up that were discovered too late to be fixed, are inherent to wavefront scheduling, and/or would have become projects in their own right to solve. These issues include:

1. Infinite loops when applying O1,O2,O3,or O4 with wavefront scheduling
2. Applying wavefront scheduling when LLVM is running in multithreaded mode
3. Applying wavefront scheduling to extremely large programs.

5.2.1 Failures associated with applying O1,O2,O3, or O4 with wavefront scheduling

Currently, it is not possible for wavefront scheduling to complete if O1,O2,O3, or O4 (LTO) is applied prior to or concurrently. This is due to a bug in how non-local variables are handled by the implementation which makes the incorrect assumption that all non-local variables are constants. Fixing this issue is not difficult, but was found late in the testing process. Almost all of the testing performed during implementation was done using unoptimized versions of the test programs. This was done for two reasons. First, it was found that there was a lot more scheduling potential in the un-optimized version compared to the O1, O2, O3, and O4 versions. And second, the wavefront scheduling pass will infinite loop on most code when O1,O2,O3,or O4 has been previously or concurrently appleid. However, it is totally safe to apply wavefront scheduling and then O1,O2,O3, or O4 as a second *separate* optimization pass. This is why it is possible to have the results of O1,O2,and O3 shown in the testing section.

5.2.2 Multithreaded LLVM and CLIPS

In the advanced programming guide for CLIPS, it is mentioned that CLIPS is inherently single threaded and is not designed to handle multiple threads of execution within the same process[7]. This is a problem because the LLVM manual describes the ability to use threads as a way to analyze different portions of a given file in parallel. Thus it requires the introduction of synchronization mechanisms into CLIPS to allow different CLIPS environments to execute in parallel.

5.2.3 Applying Wavefront Scheduling to Extremely Large Programs

It was decided,(initially as a joke), to take an LLVM bitcode version of the open source video game *Arx Fatalis* and apply wavefront scheduling to it. The program weighs in at roughly 195 megabytes as dissassembled bitcode. In addition to taking nearly a minute and a half to load the application, all 32 gigabytes of the development machine's RAM was consumed in the next 90-120 seconds! It is believed that this is due to the fact that the build was compiled under what is known as a unity build. This takes all of the source code that makes up the program and dumps it into a single file. This takes longer to compile but more optimizations can be applied because all of the code is a single file. It would be ideal to apply wavefront scheduling to the entirety of the Arx Fatalis source code while keeping the memory consumption below 32 gigabytes of RAM. However, extensive research is still required to figure out how to make the implementation efficient enough to handle such a large program. Even with that being said, the amount of improvement possible will be tempered by the fact that the optimization is NP-Complete.

5.3 Future Improvements

There are a wide number of future changes that could be applied to improve the code quality of the wavefront scheduling optimization. This includes scheduling loops, detecting and scheduling cycles; allowing call instructions to be scheduled in limited cases, in-depth memory analysis, and changing how regions are formed.

5.3.1 Scheduling Loops

One of the limitations imposed in the current implementation of wavefront scheduling is that the contents of loops are not scheduled. This is due to the fact that a suitable technique was not found during implementation to handle the fact that a loop cycles back on itself. However, during testing, a solution to this problem was found. It was influenced by a sector trick used the id Tech 1 engine to allow for deep water. A path through a region such as a loop already contains the list of elements that is has traversed through to get to this point. Identifying a back edge is as simple as checking to see if the next element is already in the path. When this situation is encountered, the path is closed with the exit block being marked as the loop itself. This creates a linear path through the loop that terminates with the ”exit block” being the next iteration of the loop. This prevents infinite loops from forming, and is a more elegant technique than ZOT [1]. Once the paths of the loop have been determined, then wavefront scheduling can be applied as though it is just another region. The only down side to this technique will be an increase in overall analysis time and memory consumed. However, the potential performance gains are large enough to make that a small price to pay. This technique was not implemented due to time constraints but will be one of the first things to be implemented once this thesis is complete.

[1]ZOT executes a loop zero, one, or two times to figure out the content of the loop.

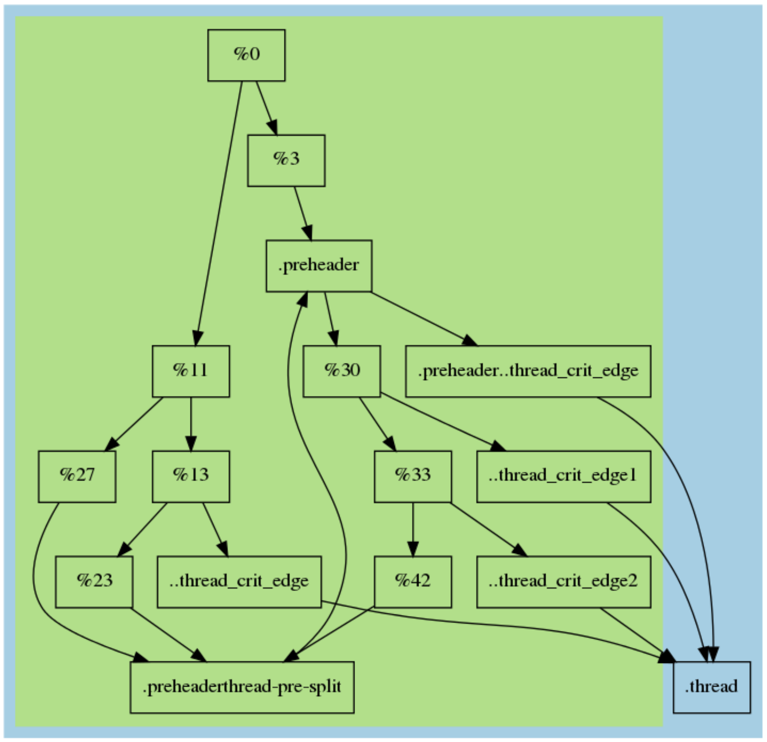
%0
%3
.preheader
%11
%30
.preheader..thread_crit_edge
%27
%13
%33
..thread_crit_edge1
%23
..thread_crit_edge
%42
..thread_crit_edge2
.preheaderthread-pre-split
.thread
Region Graph for 'GetNextPatternEntity' function

Figure 5.1: This function has had link time optimizations applied to it. Notice that while there is a back edge it is not a loop because there is no entrance that dominates all nodes that are part of the cycle. It is possible to get to the preheader from the backedge without going through the preheader.

5.3.2 Cycle Detection and Scheduling

One of the biggest flaws in the current implementation is its reliance on LLVM to detect loops. In most cases this is not a problem unless extreme optimizations are applied that convert a loop into a cycle. When this happens, LLVM will not tell CLIPS that there is a loop and the optimization will infinite loop. Solving this issue requires using the technique previously mentioned for loops but with a slight change. Instead of referencing the loop as the exit, the duplicate basic block becomes the exit. This means that the length of the path will become quite large but the path will be acyclic, making it a perfect candidate for wavefront scheduling. An example of such a cycle is defined in figure 5.3.2. As with loop scheduling, this solution will not be implemented until after the thesis is complete.

5.3.3 Allowing limited call instruction scheduling

Currently, it is not possible to schedule call instructions out of the basic block in which they originate. Because of this, a large amount of ”scheduling material” is lost to what could be considered an artificial limitation. The problem is determining which call instructions are allowed and which are not. One idea that is being explored is to fix call instructions but allow instructions that existed below the call to be moved if the instruction is not a store or load instruction. Since LLVM is register based, it is safe to assume that registers will not be modified by a call that modifies memory in an unknown way. Only memory is considered unsafe and thus load and store ordering must be considered. Further research is necessary and will be explored after the completion of this thesis.

5.3.4 In-Depth Memory Analysis

One of the biggest issues surrounding call instructions is not knowing what is modified if the call instruction has side effects or writes to memory. Therefore it is necessary to implement a form of in-depth memory analysis that actually determines what is modified by such a function. Currently, the memory analysis performed is quite limited and only resolves one depth of pointer computation. Eventually, an infinitely deep pointer analyzer will need to be constructed to resolve exactly what is being modified. Unfortunately, developing this analyzer was beyond the scope and time constraints of this thesis but if it could be developed and implemented, it would allow more instructions to be scheduled out of blocks below a call instruction. Further analysis and experimentation is required to see if this will yield any positive benefits.

5.3.5 Modifying the region selector

The paper on wavefront scheduling describes a separate region selector to construct so called *scheduling regions*. These scheduling regions consist of a small number of blocks that can have multiple entrances and exits. Where as the current implementation of wavefront scheduling relies on LLVM’s region selector which is less optimal because it will only select regions that are single entrance and single exit. If the function’s CFG does not abide by this requirement then the number of sub regions within the function will be reduced greatly. One surefire way to improve the efficiency of the wavefront scheduler is to modify the region selector so the size and shape of the region is sufficiently large but not too large. The paper notes that most of its scheduling regions are normally around seven blocks in size [4]. However, testing has shown that wavefront scheduling can also benefit from really long paths as well. The only problem being that the amount of memory consumed increases greatly with each added element on the path[2]. Further research is required to determine the best layout for region selection. However, it is clear that LLVM's region selector does require some more work. Ideally, a global scheduler would be the best way to go. However, that may not be totally possible without further research into the NP-complete aspects of the problem.

While there are other improvements that could be applied, the changes listed here are ones that are thought to be able to provide an appreciable improvement in code quality.

[2]Remember this is an NP-Complete optimization.

Chapter 6

Conclusions

This thesis has been an interesting journey in that it actually consists of two different projects. The first being integrating CLIPS into LLVM to allow expert systems to be used in compiler optimizations. The second is to study the performance impact region scheduling algorithms have on superscalar architectures.

Implementing wavefront scheduling as an expert system was the easiest way to represent the algorithm and make it maintainable. The unordered nature of the expert system also made it really easy to just write the algorithm and not worry about the order of application.

CLIPS

The use of CLIPS came with its share of issues. The biggest one was how the garbage collector would choke if it had to clean up six gigabytes of garbage data. It would actually consume more memory and then infinite loop the same *stage* of the optimization. This issue has not been reported to the CLIPS team yet as the workaround to this problem not only fixed the issue but also improved the running time of the optimization itself while consuming far less memory. The second issue had to do with a bug in how multifields are matched in the conditional section of a given rule. While the CLIPS manual states that complex pattern matches will consume a lot of RAM, it does not mention that the pattern match $?a ?last will be invalid because ?last will be part of $?a as well as an individual element. The solution to this problem is to add a $? after ?last ($?a ?last $?) which will still be correct provided that more pattern entities are provided to constrain ?last to actually represent the last element of the multifield.

The third issue with CLIPS has to do with pattern matching in general. The CLIPS manual states that complex pattern matches on multifields will slow down execution and cause an increase in RAM consumption. The solution to this potential problem is to write a given multifield pattern match to be only as complex as it needs to be.

The final issue with CLIPS had to do with object naming. While CLIPS provides a built in name field to match against, it is not actually mentioned in the users guide. This led to the creation of a common TaggedObject type that, when instanced, would acquire the corresponding instance name of the given object and store it as a symbol instead of an instance name or address. This ID field is used through out the wavefront scheduling implementation and has a side effect that is makes it more useful than the special name field. Since the ID field is a standard slot the object associated with it will not be consumed by a match. This allows objects to not only be matched more than once in a given phase but also makes it really efficient to store references in other objects. Internally, CLIPS uses a symbol table to represent all symbols it has seen. Symbols that are the same refer to the same entry in the table.

The use of CLIPS does provide some overhead but there are many planned improvements that should reduce the overhead of applying wavefront scheduling using CLIPS.

LLVM

There are three issues that cropped up during the implementation of the LLVM aspect of wavefront scheduling.

1. Ownership of exit nodes to an LLVM region.
2. Pointer Size
3. malloc versus calloc

Regions in LLVM do not actually own their exit nodes. This caused no end of trouble until realized because it was expected that the exit node of a region would be added to the path. Handling this issue required an extra check to be put in place to detect if the next element of a given path was an exit block or not.

The second major issue encountered had to do with pointer sizes. Since CLIPS interacts with LLVM it was necessary to be able to save the pointer address of the LLVM object regardless of the architecture. Since the majority of the KCE and LLVM aspects were developed on x86-64 machines it was assumed that int longs would be perfect to represent 64-bit pointers. This space would be large enough to also store 32-bit pointers by virtue of size. However, when some of the testing shifted over to an Itanium based machine the program would crash with a segmentation fault because there was not enough space to store an 64-bit Itanium pointers in a long int. This required pointers to be stored in long longs instead. This was not an issue for CLIPS because it already operates on long longs internally. Once this change was made the optimization would not crash with a segmentation fault[1].

The third major issue that was encountered was the difference between malloc and calloc. Normally, a C program dynamically allocates a block of memory using malloc. However, when allocating space for C strings it was discovered that consuming all of the system RAM would cause subsequent invocations of the optimization to crash during the KCE with strange parsing errors. It turns out that when a block of memory is freed in C it is not zeroed out. If the space allocated by malloc is larger than the actual length of the string it is possible to have garbage text appended to the end of the string. It is possible for this garbage to be parsed and crash the optimization. Fixing this issue requires the use of the calloc command. The calloc command is a wrapper over malloc that replaces all elements in the memory section with zeros to clear it out. The calloc command is necessary if one is manipulating strings in C.

While there were other issues in LLVM, they were minor ones that were cleared up after some work in GDB or reading the LLVM user manual. Even with the issues encountered, LLVM was found to be a very easy to use and extensible framework.

Implementing wavefront scheduling in LLVM and CLIPS was an interesting move as the author now has intimate knowledge of the inner workings of both. However, the overhead

[1]LLVM is not supported on Itanium so it was only used to make sure that the C++ code worked

associated with CLIPS may not be acceptable for some.

Implementing wavefront scheduling was at times a huge chore because it was necessary to *translate* what [4] meant. Other times it was necessary to fill in the blanks left by [4] as well. Even with that being said wavefront scheduling is an effective region scheduling algorithm that can be applied superscalar architectures to great effect.

This thesis has made several major contributions including:

1. Provided a mechanism to convert LLVM data into CLIPS knowledge automatically

2. Implemented wavefront scheduling for LLVM

3. Showing that an expert system can be used to optimize code

4. Showed that Wavefront scheduling improves performance on superscalar processors.

[4] showed that wavefront scheduling does provide a performance improvement when applied to code targeting the Itanium architecture. This document proves that wavefront scheduling also provides a performance improvement when applied to superscalar processors.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing - a VLIW approach to architecture, compilers, and tools*. Morgan Kaufmann, 2005.

[3] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *Micro, IEEE*, 23(2):44 – 55, march-april 2003.

[4] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront scheduling: path based data representation and scheduling of subgraphs. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 262 –271, 1999.

[5] Joseph C. Giarrantano. *CLIPS User's Guide*. Self Published, December 2007.

[6] CLIPS Team. *CLIPS Basic Programming Guide*. Self Published, March 2008.

[7] CLIPS Team. *CLIPS Advanced Programming Guide*. Self Published, March 2008.

[8] Al Aburto. flops.c, December 1992.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[10] S. Winkel. Optimal versus heuristic global code scheduling. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 43 –55, dec. 2007.