SPAWN POINT PREDICTION FOR A POLYFLOW PROCESSOR

BY

TODD M. RAFACZ

B.S., University of Illinois at Urbana-Champaign, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Because of technology restraints and limited amounts of ILP, increased single-threaded performance through automatic thread generation has been one of the more appealing approaches for advancement in recent years. Dynamic Multithreading was one of the original hardware-only approaches in this area, but it only exploited very simple control independence, among other shortcomings. Polyflow, a recently proposed architecture by Matt Frank and Sanjay Patel, tackles the problem of executing multiple threads in a much more elegant way, but with the need for better control independence information than DMT. This thesis analyzes a control independence predictor recently proposed, and extends it for the specific needs of polyflow. Also, improvements are made to greatly reduce the complexity, transistors, and power used in such a predictor.

ACKNOWLEDGMENTS

I would like to thank the many people who have helped me through my college career. The University of Illinois and its faculty have provided a great environment for me to advance my skills. Specifically, both Prof. Sanjay Patel and Prof. Matt Frank have given me the opportunity to do highly interesting work, providing guidance regarding my research and my career. I would also like to thank the students I have worked with closely: Nick Wang, Brian Fahs, Kevin Woley, Kschitiz Malik, and Sam Stone.

Finally, I would like to thank my parents for the support and guidance they have provided throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Over the last few decades, manufacturing improvements have provided both an increased number of transisters and a higher transistor speed to architects for use in microprocessor design. Large performance gains have been achieved through many things, mainly from frequency increases through pipelining at a small cost to IPC, and through IPC gains achieved through spending the increasing number of transistors available to architects. More recently, however, these gains have been slowly decreasing for many reasons, one of which is a lack of further ILP in programs to exploit.

This lack of ILP comes from two sources. One reason is that there is simply not much parallelism to be gained within achievable window sizes from current technologies. The second reason, which is intertwined with the first, is that branch prediction performance limits any performance improvements from larger window sizes.

1.1 Effective Fetch Bandwidth

Effective Fetch Bandwidth is a metric which measures the effect of branch prediction on a modern pipelined machine. Since instructions are fetched well before they are executed, a prediction must be made when a branch instruction is fetched because the outcome of the branch is not yet known. When the processor predicts the wrong direction, this is called fetching down the wrong path of a program. This is repaired when the branch that was predicted incorrectly is executed. The machine is then flushed of all the wrong path instructions, and resets to fetching along the correct path again.

The average number of instructions that can be fetched before reaching the wrong path of a program with today's branch predictors is limited. The amount of parallelism that can be exploited by a superscalar processor is limited by this fact. As machine's instruction windows become larger, it becomes increasingly difficult to fetch a large percentage of instructions along the correct path. With a single fetch point there are diminishing returns to increasing the window size because of this. One way to alleviate this problem is to have multiple points in the machine where fetch was known to start on the correct path.

Simultaneous Mutithreading [1], [2] achieves this by executing multiple OS threads at the same time, achieving a higher effective fetch bandwidth in addition to there being more achieved parallelism in the pipeline because of the explicit parallelism of multiple independent threads. Unfortunately, this approach cannot be used to increase single-threaded application performance. There have been attempts to write compilers that automatically generate theads for an application, but this has been largely unsuccessful so far.

Dynamic Multithreading [3] is one architecture which generated control independent threads through two common program constructs: loops and function calls. Their architecture used the ease of predicting the exit point of a loop and the return point of a function call to create threads that are small chunks of a program. These threads can then be pieced together to form the complete program flow. This method does help increase the average correct path instructions fetched within a machine window, but it neglects some common control independence points. Also, their proposed architecture did nothing to handle data dependences between the generated threads of the program.

Speculative multithreading [4], [5] is another architecture proposed to achieve automatic thread generation of a single-threaded program. This architecture does a good analysis of data dependences between threads of execution, but is only targeted to exploit parallelism available in multiple iterations of the same loop.

Another proposed architecture deemed Skipper [6], is an architecture built to exploit general control independence, but made no attempt to handle data dependences between the different chunks of code.

The polyflow architecture geralizes thread generation to all control independence, in addition to handling depdendences across spawn points through prediction mechanisms. The polyflow architecture proposed splits a program into many control independent chunks that are not dependent on branch prediction, thus relieving the limits of average correct path instructions that can be fetched in a processor at a time. This also helps expose parallelism that a superscalar processor is not able to exploit [7]. In addition to function and loop control independence points, it is able to exploit control indepdence from common program constructions like an if-then-else. When the “if” statement is reached, it is unclear whether the “then” or “else” portion should be executed, but it is known that the code immediately after the if-then-else should be executed, barring any return statements in either block.

In addition to better control independence exploitation, the polyflow architecture offers a few other advances which will be discussed in Chapter 2.

CHAPTER 2

POLYFLOW

This section discusses the principles of Polyflow in detail, motivating the need for a better reconvergence prediction algorithm. First, an example snippet of a program will be shown that exibits behavior which would allow a Polyflow processor to perform much better than a standard superscalar processor.

2.1 Motivating Example

The control flow graph in Figure 2.1 is a perfect example of code which the Polyflow processor excels at executing. The example is a doubly nested loop with an if-then-else statement inside the inner loop. For the sake of this example we will assume the outcome of the if branch is hard to determine.

A superscalar would execute this code sequentially, progressing one block at a time, making a prediction at each branch. The loop branch is easily predictable as taken, but the if-the-else branch is not. This will cause the superscalar to stop making progress on the program when it makes a false prediction until the mispredicted branch is executed.

Figure 2.2 shows the progression of how a polyflow machine would fetch this program section. Block A is the beginning of an outer loop iteration. From this point we know that Blocks B and E are control independent and can thus be spawned off. We will predict the branch at the end of E is taken, but also spawn off Block F because it is the exit point of the loop, which is also control independent. From Block F, we can fetch Block A again and start the unfolding of the control flow again. One of the most important takeaways from this example is that outer-loop parallelism

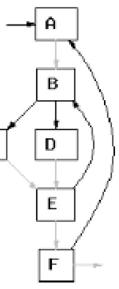


Figure 2.1: Example Control Flow Graph

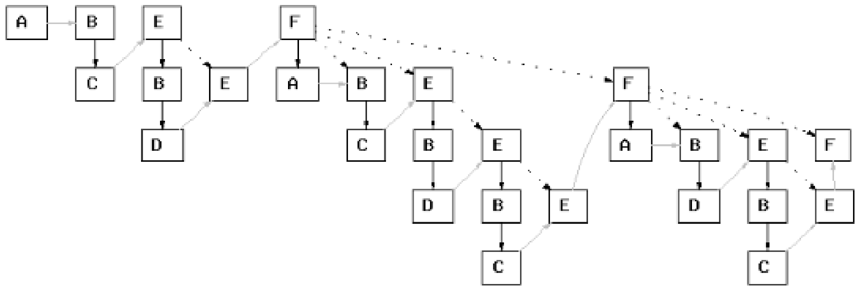


Figure 2.2: Example of Polylow's Execution

can be gained very easily through this architecture. This is parallelism that is seen between two iterations of the outer loop. A supserscalar would not be able to achieve this kind of parallelism because of mispredictions at the if statement or at the loop ending branch of the inner loop.

2.2 Definition of Terms

There are many terms used in the rest of this thesis that have been defined specifically for use in describing the Polyflow architecture. There are also a number of terms from compiler analysis that are used.

Control Independence - This term refers to a future point in a program that will execute no matter what the outcome of any branch is. Thus, is it *control independent* from the current point in the program. This is the main conecept used to find new flows to spawn, since we know that section of code will eventually be executed. The notion of control independence can be calculated in the compiler by finding the reverse dominators of all basic blocks.

Flow - A flow is very similar to a thread, except that it is produced by the machine hardware instead of a compiler. Flows are only seen by the microarchitecture of a processor, not by the OS. Flows are chunks of a sequential program that can be pieced together to form the entire program.

Spawn - A spawn is when a branch is fetched, and a prediction mechanism determines that there is a point further ahead in the program that is Control Independent. When this happens, a new flow is created that starts at the PC of the Control Independence Point.

Reconnection - This is the attempt of the processor to piece together two flows of execution to form one flow that follows the predicted execution path.

Reconnection PC - This is the PC that was predicted as a control independence point from a branch, and is the PC that a spawned flow will begin at. When the flow that performed the spawn reaches this PC, it is now eligible for reconnection at arrival.

Nonspeculative Flow - This refers to the flow in the machine that is the current thread the reorder buffer is retiring from. There is no instruction in the program earlier in time that had not been retired yet. Thus, it is *nonspeculative*.

Speculative Flow - This refers to a flow that is created by a spawn from another Flow. Any flow that is not the Nonspeculative Flow is considered a Speculative Flow.

Least-Speculative Flow - This refers to the flow in the machine that was most recently spawned by the Nonspeculative Flow.

Most-Speculative Flow - This refers to the flow in the machine that is furthest ahead in time in terms of the sequential program order.

2.3 Polyflow Architecture Components

There are a number of additions to a traditional processor that are necessary to make Polyflow work. The following sections discuss each of these at a high level to explain the foundations of correct execution of a Polyflow processor.

2.3.1 Fetch policy and spawn prediction

The front end of a Polyflow processor needs to handle much more information than does a single-threaded processor. In addition to traditional branch prediction, the front end now needs to keep track of multiple flows of execution, and predict points of independence to create new flows.

The task of predicting spawn points is handled by a reconvergence predictor. This predictor is required to predict the point of control independence at any branch. Correct execution of a program does not depend on a correct prediction of control independence, but there is a performance penalty for making bad predictions. The subject of how to predict control independence points is the main subject of this thesis. In addition to predicting reconvergence PCs, there must also be a prediction of which register and memory values may be produced by the section of code being skipped over. This prediction is fairly easy, and is based on previous instances of the code to be skipped.

In addition to finding points of control independence, a nontrivial algorithm for choosing whether or not to spawn must be followed. In the case of limited flows in the machine, one flow must be commonly be killed to make room for a new flow. Some flows also are more likely to produce good work than others. There must be some balance between limiting the number of flows killed, and not missing "good" spawn points because our machine already has the maximum number of threads. This algorithm is the subject of future work.

2.3.2 Diverter

Since the front end predicts which register and memory values may be produced by a section of code that is spawned past, we are not able to rename these registers when the spawned flow reads them. The diverter keeps track of the set of registers that cannot be renamed for each flow, and when an instruction tries to source from one of these registers or a load which is predicted to alias reaches the diverter, its destination is renamed, and is put into a queue to wait until its sources can be renamed. By allocating this instructions destination register, further dependent instructions dont need to be diverted, keeping the size of the divert queue minimal. When two flows eventually reconnect, the necessary values which the second segment was dependent on are passed to the diverter so the instructions can finally be renamed and inserted into the scoreboard.

2.3.3 Linked list reorder buffer

For optimal flow balancing, a unified linked-list reorder buffer is necessary in a Polyflow processor. It is possible to build a seperate Reorder buffer for each flow, but in cases when there is not perfect balancing of the size of each flow, there is a waste of resources.

Another main requirement of a reorder buffer is that it be able to do preretirement of instructions in speculative flows. Preretirement is when instructions in the speculative thread have executed, but cannot be fully retired because the spawn point it started at is still speculative. To save space in the rob, these instructions can be collapsed to only hold the final register and memory commit state of a large number of instructions. This frees up resource space in the reorder buffer for most flows to make progress.

2.3.4 Context manager

Finally, a logical unit is necessary to manage the creation and reconnection of flows. A number of things must be done to check the correctness of a flow reconnection. The set of registers which were read by the speculative must be checked against the set of registers which were produced by the originator thread between the spawn point and reconnection point. If any register read by the speculative thread was written by the originating thead after the spawn point, there was a bad register value used in the speculative thread, and the reconnection is unsucessful. The speculative thread should then be killed, and the originating thread would contine from the point of failed reconnection.

It should be noted that reconnection can happen at arrival (when the final instruction to join the two flows is fetched) and also at retirement (when this final instruction in the first flow is retired). In the case of reconnection at retirement, only the nonspeculative thread and least-speculative thread can be reconnected at any time. This simplifies the reconnection logic greatly. In the case of reconnection at arrival, two speculative threads can be reconnected. This complicates the process because at arrival it is unknown if any flows prior to these two have written a register that was read by the two joined flows. This check must be done later. It is done by combining the set of registers upwards exposed by the two flows, and saving this set until later. When the now joined two flows reconnect with the previous flow, the saved upwards exposed is set against the set of registers that were written by the previous flow.

This is obviously a more complicated algorithm, but the benefit of reconnecting at arrival is that dependences between two flows can be resolved much earlier now. This hopefully prevents the dataflow from the two flows from being on the critical path of retirement.

CHAPTER 3

CONTROL INDEPENDENCE

One of the main requirements of a Polyflow processor is that the hardware be able to determine control-independent points in the program to be able to spawn new flows that start in the future of the program. There are a number of ways this can be done, but they all fall into two categories of static and dynamic analysis.

3.1 Static Analysis

Static analysis can be defined as information about a program that can be gained without observing any sample executions of the program. This analysis can be done by the compiler before the binary is created, or after compilation by examining the binary.

3.1.1 Compiler analysis

It is relatively simple for a program to analyze the control-independent point for every branch in the program. This information is commonly generated already to perform various optimizations [8]. It can be calculated by generating the Reverse Dominators of all basic blocks of a program. This is one of the better sources of control independence for the creation of spawns, but the main issue is how to get the information from the compiler to the processor. Some information can be sent through the binary, but generally there would be too much information because the compiler would have a hard time filtering out the useful spawn points at compile-time. Also, using the compiler is bad because it would be necessary to create new binaries for all existing software. The length of time X86 has survived as one of the main instruction sets is a testament to how important backwards compatibility is.

3.1.2 Binary analysis

Another technique that has been used in the past is to determine control independence points from common program constructs that can be identified from the binary. Dynamic Multithreading (DMT) [3] used the two constructs of loops and function calls to find control independence. A function call's control independence point will always be the instruction immediately following the function call. In rare cases this is not correct, but does not happen enough to be a factor. DMT identified loops by backwards branches, or branches whose taken PC is less than the PC of the branch. Because of the way compilers create loops, this captured a large portion of loop behavior.

There are a number of reasons why this technique would not work in Polyflow, however. First of all, DMT spawned control-independent threads to difference cores, which did not affect the performance of the nonspeculative thread. In polyflow, an SMT machine is used, which shares all resources between flows. In this case, it is clearly necessary that we not spawn a new flow unless we are relatively sure it is useful and will reconnect within a relatively short amount of time. This will be explored further in the sections about the reconvergence predictor.

3.2 Dynamic Analysis

Dynamic analysis is an analysis that is done by viewing execution paths of a program. This is deemed dynamic because the information is always changing, but generally this information is the best for predicting program behavior in the near future.

There are two main benefits to using dynamic analysis for predicting control independence points. The primary reason is that there would be no required change in the instruction set to build a Polyflow processor. This is a very important aspect if this architecture were to ever gain traction. In addition, transmitting the information through the binary may be inefficient if the compiler or profiler were unable to filter out unnecessary spawn points.

The second benefit is that dynamic information is generally better. There are many branches in programs that are statically not taken, or statically taken. This behavior can cause static analysis to be much more conservative than necessary, and can cause some spawn points to commonly reconverge after one instruction. A common example of this is an assert statement in a function. As the nature of an assert, it is rarely, if ever executed, and when executed it is the last part of execution of the program. If an assert were to be inside an if-then-else construct, a compiler would be unable to choose a control independence point for the branch corresponding to the "if." This is because the program would exit if the assert were ever executed, causing the control independence point of the if-then-else to never be reached. However, with dynamic information we could see that this assert is never executed, and choose the correct spawn point. This behavior can also be seen from both the break statement in C, and in return statements from within loops. Break and return statements that are rarely executed can cause the spawn points of some branches to be overly conservative.

The need to avoid tracking uncommon behavior like some break and return statements can be seen from the results in Chapter 6. They discuss the performance benefit from limiting the number of branches that we record behavior from at a time, and from the benefit of occasionally clearing our reconvergence predictor.

CHAPTER 4

DYNAMIC RECONVERGENCE PREDICTION

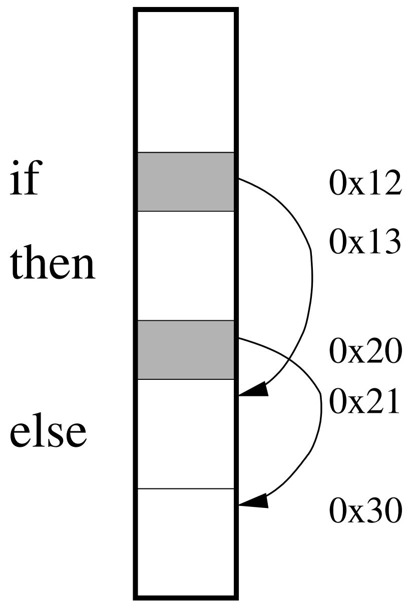
This chapter will discuss the basic algorithm proposed for reconvergence prediction proposed in [9]. This dynamic reconvergence predictor was very good at predicting a point of reconvergence, or control independence, for any branch. In order to describe the changes necessary to make to the predictor, first the algorithm will be described at a high level.

The Reconvergence Predictor trains itself with information from the retired instruction stream produced by the Reorder buffer. It then uses this information to predict points of reconvergence when a branch is fetched. The predictor works by allocating an entry in a table for each branch retired. This entry holds three possible reconvergence PCs, in addition to other control information. These PCs are the *reconverge below*, *reconverge above*, and *rebound reconverge* entries. These PCs cover most behavior of simple program constructs.

The reconverge below pc captures the behavior of all code that reconverges below the branch, without jumping to a point below this PC before reconvergence. The reconverge above PC captures reconvergence points above the branch, and the rebound reconvergence PC captures reconvergence points that are more complicated, where code is executed both below and above the branch before reconvergence.

4.1 Update Algorithm

The update policy of the reconverge below PC will now be explained in more detail. The reconverge below PC is initialized to the PC immediately after the branch. The update policy is that the entry



iteration	direction	old	new
1	nt	0x13	0x13
2	t	0x13	0x21
3	t	0x21	0x21
4	nt	0x21	0x30

Figure 4.1: Algorithm Steps for Reconvergence Predictor

will remain open until either the current predicted PC is reached, or a new PC that is greater than the current predicted is reached. If a new PC greater than the currently predicted PC is reached, the new predicted PC is set to this value. Once the predicted PC is seen or updated, the entry is closed until the originating branch is seen again.

This algorithm can best be understood through an example of an if-then-else statement as seen in Figure 4.1. Branches and jumps are indicated in the code section by a shaded region, and important PCs of the code segment are labeled to the right. This example details the update of the reconverge below PC, though it should be noted that the same updates to the other two reconnection PCs are done in parallel.

When the branch is seen for the first time, the reconverge below PC is initialized to the PC directly below the branch. In this case, that PC is 0x13. In the first iteration, the branch is not taken, and the predicted reconnection pc is reached immediately. Since we reached the predicted PC, we close the entry until we see the branch again.

When the branch is seen again, the entry is opened, and the predicted PC is 0x13 again. However, the branch is taken this time, and the next PC seen is 0x21. Since this is greater than 0x13, the predicted value is updated to 0x21 and the entry is closed.

On the third iteration of the branch, the predicted PC is 0x21, and the branch is taken. Since we immediately hit the predicted PC again, we close the entry.

On the final iteration of this example, we once again predict 0x21 as the reconnection PC. On this iteration, the branch is not taken. The code follows the “then” path, which all have PCs less than 0x21, and then when the code reaches 0x20, jumps to 0x30. Since we never hit 0x21, but eventually reached 0x30, we update the value to 0x30 and close the entry. It can easily be seen that this is the true reconnection PC for the “if” branch, since it is the first PC that is guaranteed to be seen no matter what path the program takes.

4.2 Predictor Analysis

It should be noted that the type of independence this predictor finds is slightly different than the type of information that would be provided by a compiler. Figure 4.2 shows a control flow graph that highlights the differences in spawn points from these two methods.

In this figure, program flow edges are shown as solid lines, and possible spawn points are shown as dashed lines. Lines 1 and 2 are possible spawn points from compiler analysis, while lines 3 and 4 are spawn points found through the reconvergence predictor.

The first difference highlighted by this example is that spawns occur from the beginning of a basic block from compiler analysis, and originate from the branch at the end of a block with the predictor. This can have a significant impact on spawn distances depending on the size of the block. However, the most important impact is the difference in data dependences across spawns from the two different originating locations.

In this example, if instructions in Block E were dependant on values produced in Block B, then the spawn point should be at the end of Block B. If the spawn point were at the end of the block, then when we spawn the new flow’s rename tables will know the destinations of the registers Block E is dependent on. If we chose to spawn at the top of Block B, the new flow would not have the rename information for these dependant instructions, and would have to divert these instructions until reconnection, where the updated rename table is communicated to the spawned flow.

The reconvergence predictor could easily be modified to make spawn predictions from the beginning of basic blocks, but the question of when it should do this cannot be answered without analysis using a full simulation of polyflow. This will be the subject of future work.

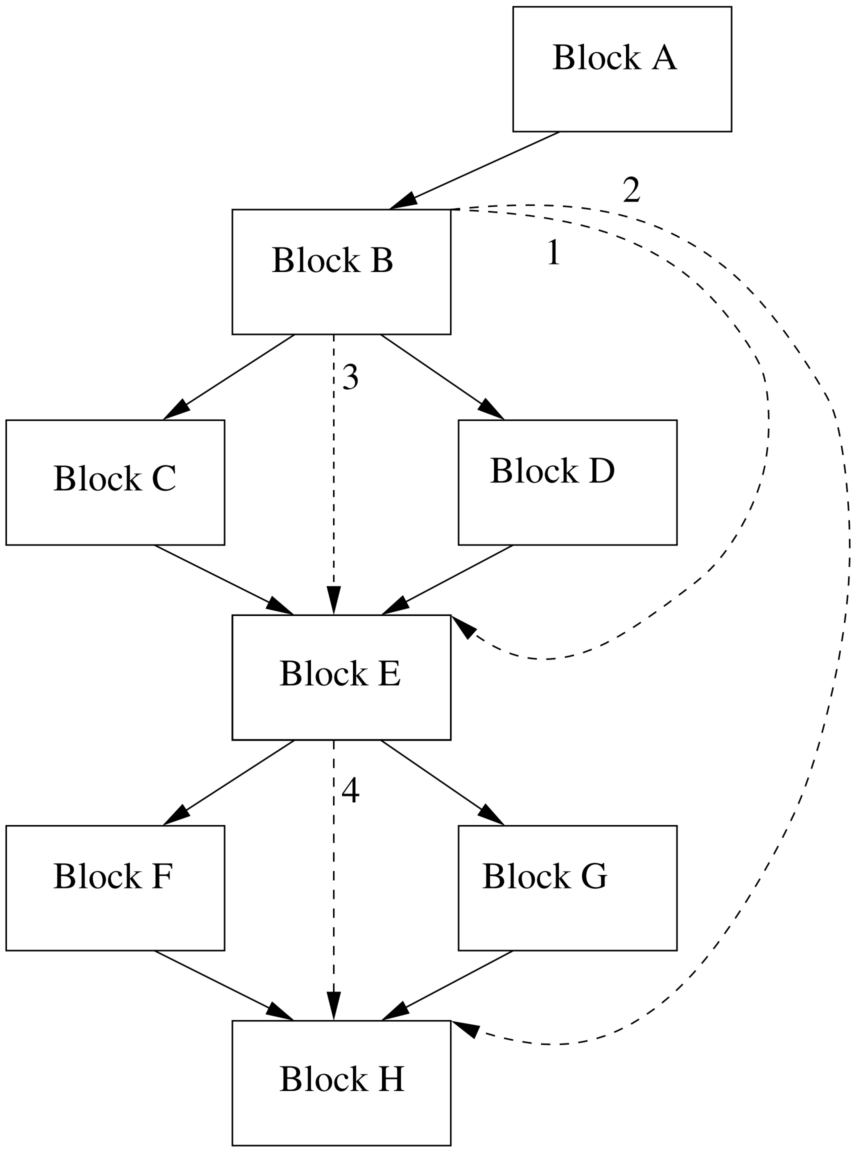


Figure 4.2: Example of Spawn Points from Both Dynamic and Static Analysis

Another important difference between the compiler and predictor spawn points is that the compiler is able to combine the smaller spawn points into one larger spawn as seen from arc 2. The ability to combine two spawn points in the predictor is discussed in Section 5.5. However, this is another situation where it is unclear which method is better for use in Polyflow. Depending on the size of the blocks, it could be better to create two spawns from the two if-then-else's, or if the blocks are small, it could be better to only use the one larger spawn. The topic of which spawn to use is also the subject of future work.

CHAPTER 5

CONTROL INDEPENDENCE PREDICTION ADAPTATIONS FOR POLYFLOW

This chapter will detail some of the changes made to the reconvergence predictor in order for it to be useful for Polyflow. There were many slight changes in the reconvergence predictor, but some of the most significant involve targeting a specific class of branches.

5.1 Methodology

Before discussing any measurements made from the reconvergence predictor, the methodology of these measurements should be described. The predictor was trained and tested by a trace of instructions produced by the Simplescalar functional simulator [10], running the Alpha instruction set. The simulator was run on the SpecInt2k benchmarks, optimized at level -O3. Reduced input sets were used to make simulation time more manageable.

5.2 Targeted Control Independence of Polyflow

For every branch in a program there is a point of reconvergence, but not all branches have spawn points which can be benefitial in polyflow. There are a few reasons why a majority of branches are not good potential spawn points. These restrictions will now be discussed, because for later studies we will measure their impact on the particular class of instructions that we are targeting.

One of the restrictions on spawn points is that they reach their reconnection point within a number of instructions that is on the order of the window size of the machine. Limited Reorder buffer compression can be done on speculative flows to alleviate the restriction on reconnection distance, but only to an extent. The Reorder buffer is able to compress the changes to the register file from pre-retirement, but all stores must be buffered for later memory dependence checking and to commit to memory. This restriction can significantly impact the number of spawn points, as every loop, function call, or branch that does not reach its reconvergence point within a number of instructions on the order of hundreds is not useful.

Another class of branches that Polyflow is unconcerned about are branches whose reconvergence point is a point which would already have been spawned by a previous branch. One common example of this type of branch is an if-then-else whose "if" condition is a logical OR or AND of multiple values. In this case, there will be a branch on both of these conditions, and they will both reconverge at the same point after the "else." We would likely spawn to the reconvergence point upon reaching the first branch. When the second branch is reached, there is no reason to spawn again because a flow starting at that point has already been created.

Other examples of these branches that point to the same reconnection point are numerous. One common example is a branch where there is a return statement along one of its paths. This branch's true control independence point is then the return statement, but we would likely have already spawned the return point of the function when we fetched the call instruction for that function. Another common example are all iterations of a loop. The loop-ending branch of a loop will have its reconnection point be the exit of the loop. Obviously, we only need to spawn the loop-exit once, not every time we see the branch. Also, if there are two branches that can exit the loop, we only want to predict the spawn point once.

Figure 5.1 shows the average number of spawn points per thousand instructions for each benchmark. The Base Spawn Points category filters out all control flow that cannot be predicted by the reconvergence prediction algorithm, filters out all static instructions, and also filters out all spawn points that reconverge at a point already spawned. As can be seen, this filters our a large fraction of branches, but still leaves a significant number of spawn points.

Another very common class of branches that are not considered in polyflow are static branches. A static branch is a branch whose direction is always the same. Branches are commonly filtered out of prediction tables until both directions have been taken, to prevent wasting space on branches that are very easily predictable. The same was done in [9] to save room in the reconvergence predictor.

In addition to wasting space, static branches are not desirable because they reconnect after one instruction, as predicted by the Reconvergence Predictor. There is no benefit to be had, as no instructions were skipped, while still taking the penalties associated with the overhead of spawning a flow. Static PCs are a subset of a larger class of instructions that reconnect within a very short distance. These will be discussed in the next section.

5.2.1 Spawn breakdown

Figure 5.2 is a breakdown of the available spawn points broken down by whether they are function calls or branches. This graph is a good motivator for reconvergence prediction on branches in addition to function calls. For some benchmarks like mcf, eon, and vortex, function call independence prediction would probably provide enough spawn points to get most of the benefit of polyflow. Other benchmarks, however, will heavily rely on control independence from branches to find flows of independent work.

5.3 Additional Information Recorded by the Reconvergence Predictor

For use in Polyflow, it is necessary to track additional information than just the reconvergence point. This additional information is what allows the predictor to better determine which branches have spawn points that are benefitial to the processor.

It is useful to note that since all of this information is needed to make a spawn decision, the reconvergence predictor allocates entries for function calls in addition to branches. The entries are not needed to make the reconvergence prediction for the function call because this is trivial, but the information described in the rest of this section is very useful.

5.3.1 Average reconvergence distance

One major statistic that helps determine what is a good spawn point is the average reconvergence distance measured in instructions. This information is recorded from previous iterations of this branch. A counter is needed to count globally how many instructions have been retired; then

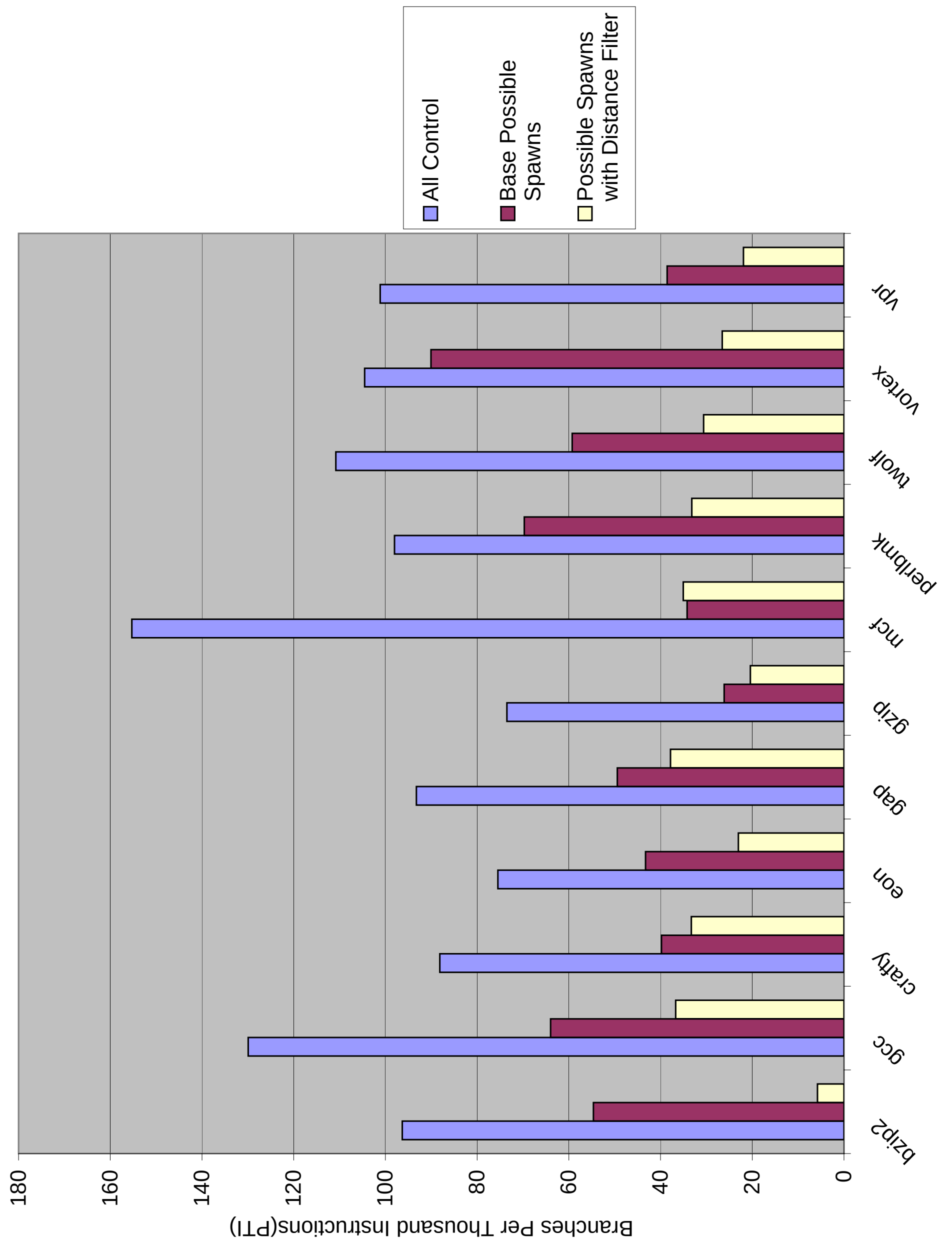


Figure 5.1: Measurement of Possible Spawn Points

all that is required is to record is the value of this counter when an entry is opened. When the reconvergence point is reached, the original counter value is subtracted from the current counter value, providing the reconvergence distance.

This value is then stored in the entry before being written into the table of entries. There are a number of ways this can be stored. The most recent value can be stored, an average of the last n values can be kept, or you can average the new value with the old average. The final option

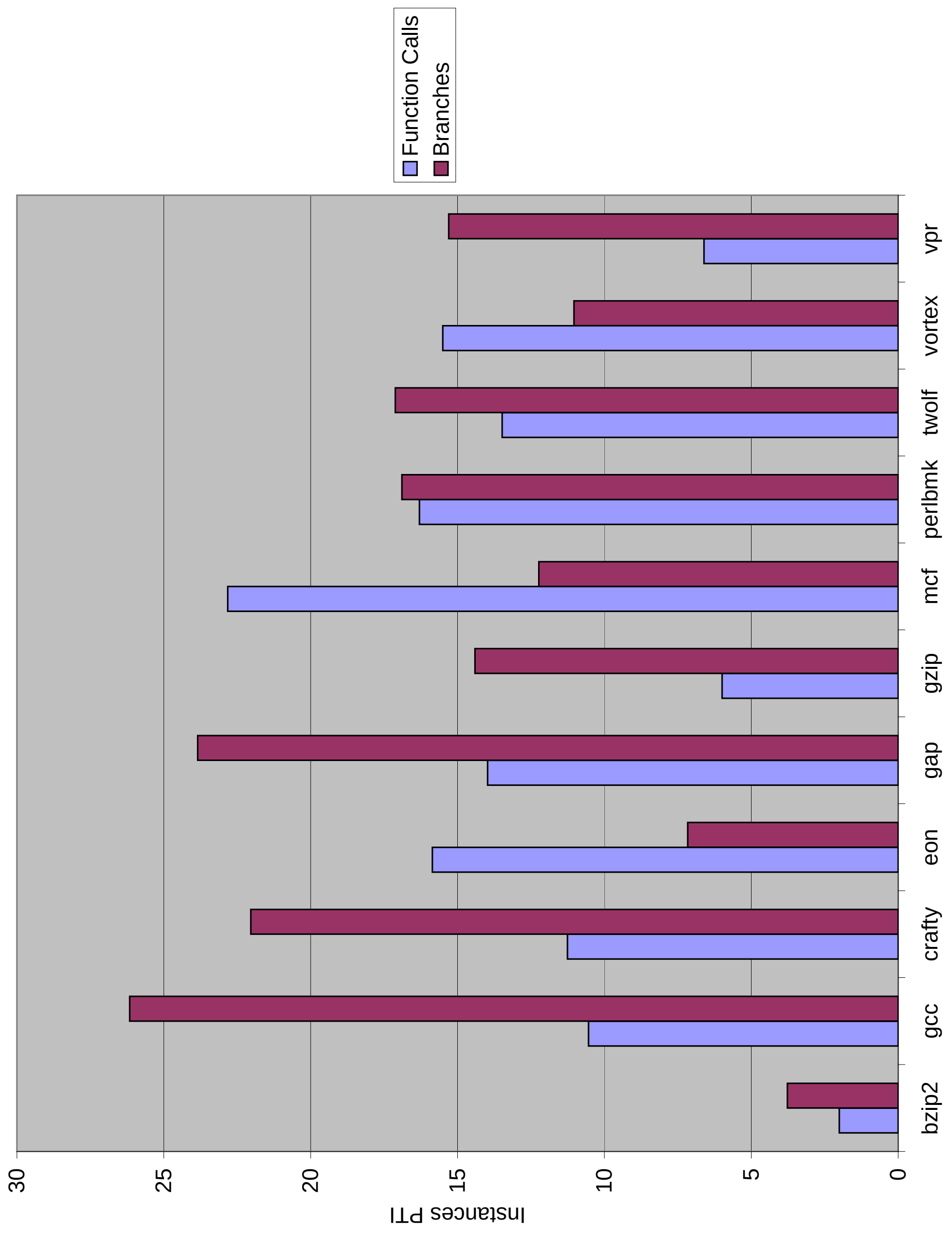


Figure 5.2: Measurement of Possible Spawn Points

would seem to be the best, as it holds information from many iterations, but favors the more recent behavior. This decision is another design aspect to be studied in the future with a model of Polyflow.

Another filter to be used on the class of branches to be considered as spawns is that its reconvergence distance on average must fall within a certain range of instructions. It was mentioned earlier how a spawn's distance must be less than a number instructions on the order of the machine window. To be conservative, that limit was set as 1000 instructions for this thesis. In addition, a branch must also reconnect on average more than two instructions away to be considered. The method of averaging the previous average with a new distance was used to calculate the average reconnection distance. In Figure 5.1, the category of Possible Spawns with Distance Filter shows how these two restrictions furter restrict the possible spawn points. Fortunately, this number is still within an acceptible range for all benchmarks besides bzip. The issue of infrequent spawn points will be discussed later in this thesis.

5.3.2 Unsafe register prediction

A very useful piece of information for use in a spawn is the vector of which registers might be written between a spawn point and reconnection. The superset of all registers that have been written between the spawn point and reconnection point is very easy to record. One bit per architectural register is needed to store this information. Whenever an instruction is retired, the bit corresponding to the destination register of the instruction is marked.

This method of recording a conservative set of registers to predict as unsafe to be renamed is very effective at ensuring correct reconnection. The accuracy of the reconvergence predictor has not been mentioned until now because it was thoroughly studied in [9]. For all studies in this paper, however, a successful prediction was measured as both a correct reconnection point, and a correct set of unsafe registers for the renamer. This only reduced the prediction rate of the reconvergence predictor by 2-3 percent, but that was decided as the metric to be used for this paper.

Results in Table 5.1 show that the reconvergence predictor predicts a correct unsafe register set very accurately. These numbers can be seen under Accuracy in the column labeled "0".

This prediction mechanism can be overly conservative many times, causing performance loss when instructions are diverted when they dont need to be. This can be addressed by a prediction mechanism that has negative feedback when some registers were predicted as unsafe but were actually safe. This issue can also be addressed by the method of clearing the entire table periodically. The effectiveness of these two methods is another study that is future work to be done with a full Polyflow model.

5.3.3 Prediction confidence

Another very significant difference between the needs of a reconvergence predictor for Polyflow, and the predictior described in [9] is that a prediction is not needed to always be made. The penalty for a bad spawn point prediction can be severe, so some sort of measurement of predictor confidence is necessary. This difference also exists between Polyflow and DMT. In DMT, a failed reconnection bears no penalty to the main core; it is only a loss of potential gain. Therefore, a measure of confidence for all predictions is necessary for a reconvergence predictor in Polyflow.

The method proposed in this thesis uses a counter for each entry in the predictor to measure how many times in a row the prediction for that branch has been correct. Upon a reconvergence misprediction, we set this counter to 0. With this counter, we can choose not to predict a reconnection until we have reached a threshold of a certain number of predictions correct in a row.

This method is effective because of the nature of the prediction being made. Once all paths between the spawn point and the reconvergence point are seen, the predictor should have the correct prediction forever. When an entry is first created, it will usually mispredict each time a new path is taken after the initial branch, depending on the hierarchy of the control flow graph associated with this branch.

As can be seen by Table 5.1, this method of only making confident predictions can increase the correct prediction rate without significantly decreasing the number of predictions made. Requiring two correct predictions before actually choosing to spawn causes the most significant gains in accuracy, with only 3 percent loss in correct predictions. Moving on to requiring eight correct predictions cuts the number of mispredictions by 75 percent, while only decreasing the number of correct predictions by another 3 percent. However, moving to requiring 64 correct mispredictions causes much more significant impact in the number of correction predictions. Therefore, requiring eight correct predictions before actually predicting a spawn appears to be the best balance in accuracy and number of spawns. This parameter will be set as the default for all further studies in this thesis. It will be discussed further, but this measure of confidence can be leveraged for other benefits as well.

Table 5.1: Comparison of Accuracy and Predictions When Varying the Required Correct Parameter

Required Correct	0	2	8	64
Accuracy	.9466	.9899	.9976	.9996
Correct Predictions	27.66	26.86	26.15	24.85

5.4 Useful Work Measurment

One useful measurement of a spawn point other than those measured already is the amount of useful work that can be done before reconnection. Because there may be many data dependences between the instructions skipped by a spawn and the instructions from the spawned thread, many instructions may need to be diverted and cannot be executed until after reconnection. A good metric of this factor for each spawn point is the number of instructions that were executed before a successful reconnection.

Other metrics of useful work that can be taken are cache misses serviced by a spawned thread, branches executed by a spawned thread, and whether or not a spawned thread was successfully reconnected. If a particular spawn point frequently needs to be killed because of store set mispredictions or because it is killed to make room for other spawns, this should be recorded to prevent it from being spawned in the future. These metrics of feedback based on execution are very useful to the context manager for use in spawn decisions.

5.5 Spawn Joining

The possibility of spawn joining was briefly mentioned in Section 4.2. Because some sections of programs may consist of many if statements with small bodies of code, there may not be many spawn points that fall within a target range of distance measured in instructions. Compiler analysis would be able to provide control-independent spawn points that jump over multiple branches, so one goal of a reconvergence predictor might be to produce this as well.

The possibility of joining two spawn concurrent spawns was studied. After reaching a reconnection PC, all spawns that reconnected waited until the next branch was reached to be closed. When this next branch is reached, its spawn point is combined with the previous spawns if both spawn points have reached the threshold of having more than eight correct predictions in a row. There is then a new combined spawn pc field to hold the spawn point, and a confidence field is kept for this combined spawn as well. When the combined spawn point is predicted correctly, its counter is incremented. When there is a misprediction, both the combined and normal counters are cleared. This prevents the predictor from attempting to combine spawns too frequently if it is unsuccessful on some spawns.

Figure 5.3 shows that the possibility of combining spawn points may have limited benefit. On average, there are only 1-2 spawn points per thousand instructions that can be combined in this way. This is caused my multiple factors. One factor is that there are too many branches that are unable to be used as spawn points. Code that has break statements and functions that have return statements other than at the end of a function are bad for independence prediction. Break statements and multiple return statements cause many more branches to have loop exit points and function returns as their reconnection point. This is a problem because function returns and loop exits can only be spawned once; additional attempts to spawn to the same point will fail.

Another factor in the lack of joined spawn points is that there may be too many simple loops in programs. A loop whose main body contains one or no if statements will have no ability to join spawn points. Upon reaching the loop branch the first time, the exit point of the loop will be spawned. Then, every time the if statement is reached inside the loop, it will try to join its spawn point with the loop branch's spawn point, but this point has already been spawned and cannot be spawned again. The two factors mentioned above cause the predictor to need some other mechanism to find larger spawn points.

5.6 Inner Loop Spawn Points

One of the main goals of attempting to join two spawn points was to be able to find inner loop parallelism. This can be defined as parallelism between two different iterations of the same loop. There will always be some dependence between two loop iterations, like the loop counter value or the pointer value followed in a list. However, this may be the only dependence in many cases. Whenever we reach a branch inside a loop, we already calculate a spawn point that is the true control independence point. However, a better spawn point may be the top of the loop body for the next iteration of the loop. While this may not actually be control independent, because the

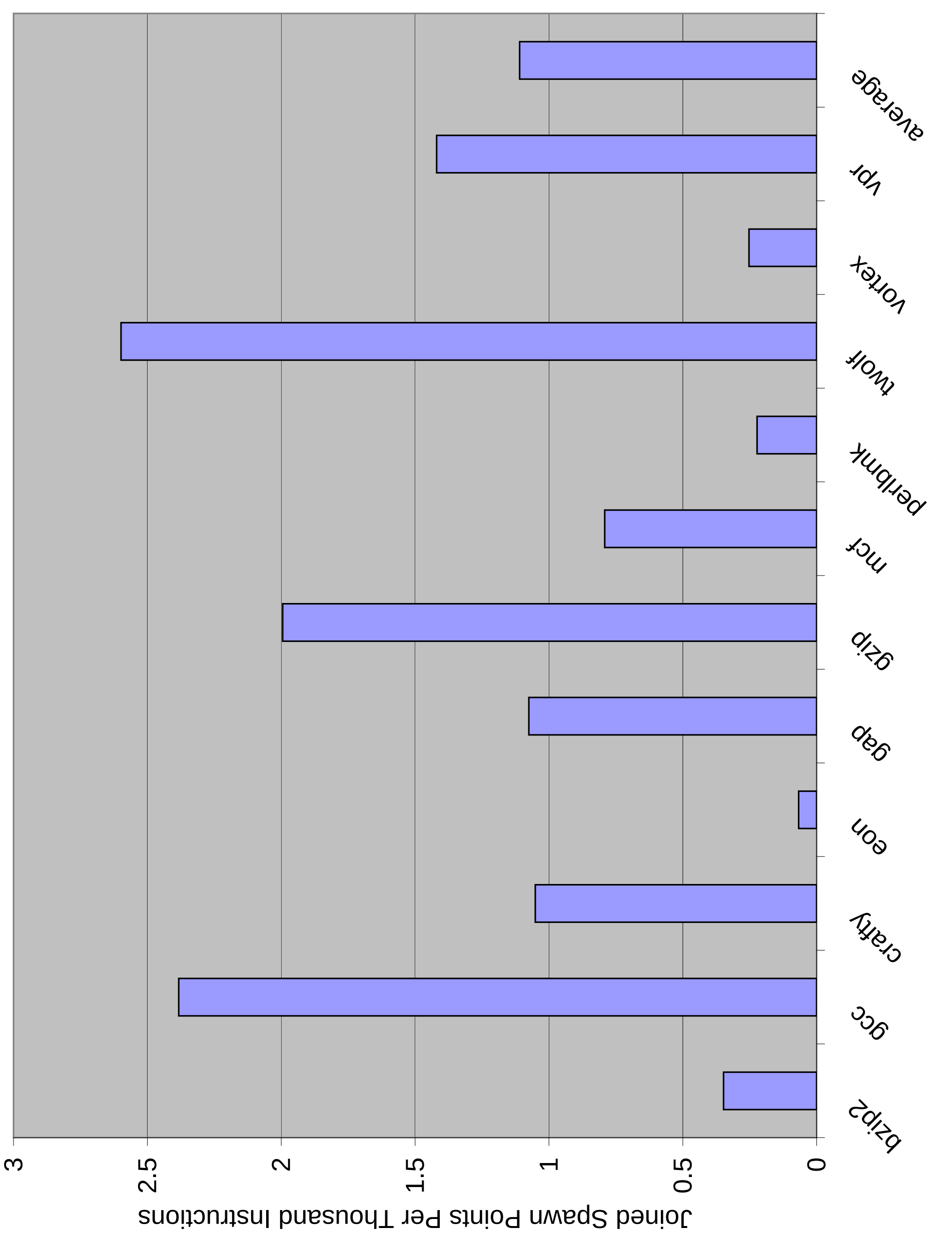


Figure 5.3: Measurement of Possible Joined Spawn Points

loop may exit before the next iteration of the loop, it will most likely be executed and might be a good spawn point.

It was mentioned briefly earlier, but the reconvergence predictor keeps an entry for a reconvergence point below the branch, above the branch, and an entry that tracks more complicated reconvergence below the branch. In the case of a simple if statement inside a loop body, the reconverge above entry actually holds the top of a loop body. In many cases, this may be a very appealing

Table 5.2: Measure of the Number of Instances to the Top of a Loop Body

	Instances PTI
bzip	5.26
gcc	6.46
crafty	1.15
eon	4.19
gap	3.36
gzip	1.91
mcf	1.44
perlbmk	1.72
twolf	11.2
vortex	3.37
vpr	0.80

spawn point, which may be able to exploit parallelism between two different loop iterations. Table 5.2 shows the average number of instances of this type per thousand instructions.

The results from Table 5.2 show that a significant number of spawns to the top of a loop are possible. These are spawn points that were not measured in the initial study on spawn points available for polyflow. While these spawn points may be coming from branches that have a reconverge below PC that is the correct reconvergence point, we now have two different spawns to create from a single branch. The most significant impact of this method of spawning is in finding spawn points in very simple loops. Loops that had no potential spawn points in them before now can fetch multiple iterations in parallel. A loop with only a static branch in it will now spawn the next iteration upon reaching this branch. Also, a loop that is only a basic block will be able to spawn the next two iterations of the loop at the same time. It may seem trivial to be able to spawn the next two iterations of a loop at the same time, but the predictor is what makes us able to distinguish a loop branch from any other branch that happens to point upwards.

The results from bzip can be considered the most significant in comparison to Figure 5.1. In the original study, bzip lacked any significant amount of spawn points available for Polyflow. This is because the benchmark is most significantly large loop bodies with little to no control flow inside. The addition of the spawn points from this method put bzip's total spawn point levels in a much more acceptable region.

5.7 Implementation Improvements

One of the major drawbacks of the reconvergence predictor as outlined in [9] is that its implementation is somewhat complicated. The RAM that stores entries for branches is a relatively simple design, but what they deem as the Active Reconvergence Table (ART) is a nontrivial design.

They propose the ART to be built with up to 32 entries per level, and requiring four or more levels of these entries. The ART operates by holding all open entries at the current function call level. When a function call is reached, the oldest level of the stack is cleared, the current entries are pushed onto the stack, and the active 32 entries are cleared to be used by this new function call level. Upon reaching a function return, any open entries are first closed, marked as hit return (predict return as the next reconvergence point), and written back into the table holding entries. Then the next highest call level is popped off the stack to continue training.

The prioritization of limited entries per stack level is done the same way as the prioritization of stack levels. When the predictor runs out of stack levels, the oldest one is sacrificed for the new stack level. In the same manner, when we run out of entries at one particular stack level, the oldest branch at that level is sacrificed for the new branch that was looked up. In cases with limited number of entries, if statements that contain several layers of hierarchy may never get their outermost branches predicted correctly. An example of this type of hierarchy is an if statement whose then and else regions contain if statements as well. Also, branches early in a function call that do not reconverge quickly might have their entries always removed from the ART, preventing it from ever being trained.

This method of prioritizing entries at a particular call level and then pushing and poping the stack is particularly poor in the case of polyflow. Because we must predict an unsafe register vector for each spawn point, we need to record the registers written during a function call between a spawn point and reconnection point. When a call is reached we cannot put the entries on a stack and stop updating the values. It would be possible to record all registers written during a function call, and return this value when we pop the stack to update the open entries, but a much more elegant solution is available that avoids the need of a stack altogether.

5.7.1 Entry prioritization

The method of prioritization originally used is similar to DMT's thread prioritization upon a new thread spawn. In DMT, when a new thread is spawned, the thread that was spawned longest ago is killed to make room for the new thread. While effective in DMT, this method is not well adapted to the type of operation that the predictor is performing. While the predictor can get better information over time, the most common behavior of the prediction for a single branch is to be updated frequently in the first few iterations of the branch, eventually find the correct reconnection PC, and then never update again. Therefore, it is highly unnecessary to continue training each entry every time we see it.

Instead, we propose a new method of prioritizing the low number of slots for open entries at any time. First of all, we eliminate the method of using a stack of entries for each call level. As mentioned above, entries need to remain open when a function call occurs before its reconnection so that it can record the registers written inside the function. This intensifies the problem of choosing which entries to remain open. To solve this problem, we prioritize the open entries by the confidence we have in each entry's prediction.

As proposed in Section 5.3.3, a confidence value for each entry in the table is kept to aid in spawn prediction. When all slots in the ART are full and a new branch is reached, the entry of the branch just reached will contend for an open slot with the entries already open. If the new branch's entry is more confident than the entries open in the ART, none of them will be evicted. If the new entry is less confident than any of the entries in the ART, the most confident entry is evicted, and the new entry will take its place.

Initially, this algorithm will favor training the smallest spawn points at the deepest level of control hierarchy. It will continue to "kick out" open entries to favor entries at deeper levels of hierarchy. Once the final level of the function stack and control hierarchy are reached, the innermost branch will get trained. Once this entry has been trained enough to be correct more than once in a row, the next level up in the hierarchy will get priority to be trained. This cascading action will traverse all levels of the hierarchy, eventually training all branches until they are correct. Once all branches have reached this state, the predictor will then cycle through all branches to increment their confidence levels. In this manner, the predictor will not train entries on all instances of a

Table 5.3: Spawn Accuracy of New ART Prioritization Algorithm

	0	1	2	8
bzip2	0.9978	0.996	0.9975	0.9978
gcc	0.9968	0.9934	0.9952	0.9965
crafty	0.9967	0.9965	0.9966	0.9967
eon	0.9993	0.9989	0.9991	0.9993
gap	0.9979	0.9973	0.9976	0.9979
gzip	0.9992	0.9992	0.9991	0.9992
mcf	0.9974	0.9973	0.9973	0.9974
perlbmk	0.9953	0.9931	0.9946	0.9952
twolf	0.9979	0.9964	0.9973	0.9979
vortex	0.9989	0.9983	0.9987	0.9989
vpr	0.997	0.9969	0.997	0.997
average	0.997654545	0.996663636	0.997272727	0.997618182

branch, but it will periodically return to training a specific branch to sample if new behavior is evident.

One issue with this algorithm is that there are some branches that may never be able to predicted correctly, or they may not reconverge for very long periods of time. This is partially solved by our initial filtering of instructions that reconverge after more than one thousand instructions. In addition to closing an entry once it has been open for too long, we also need to prevent the predictor from reopening the entry every time it is seen, because we know it is likely to fall outside the window of possible reconvergence. Allowing this class of branches to be repeatedly trained would be disastrous, as they would never be predicted correctly, allowing other entries space in the table. Additionally, because they never reconverge they take up a slot in the ART for a very long period of time.

However, we do not want to exclude these branches from training, in case they eventually reliably reconnect within the required distance. This is achieved by having another value in each entry that records whether it fell outside the window the last time we tried to train that entry. The next time we see this branch, we will not attempt to train it. Also, there is another counter recording how many times we have seen this branch since it did not reconnect inside the window. We then choose not to train this branch for a certain number of skipped attempts after it initally did not reconverge. This algorithm allows us to not waste precious ART space with entries that do not reconnect within a wanted range initially, but will check periodically if the branch actually begins to reconverge within the window. Through experimentation, a value of 16 was chosen as the number of times to skip training before allocating an ART slot for the branch again.

With these significant changes to the ART allocation algorithm, a much more compact design is possible that saves transistors and power. Also, this new algorithm is able to train the unsafe registers through function calls between the spawn point and reconnection point. Results in Table 5.3 show that this algorithm suffers no loss in prediction accuracy. This is largely due to the fact that we require eight spawn predictions to be correct in a row before allowing a true prediction. Thre real measure of the impact of this algorithm is the change in frequency of spawns. This data is represented in Table 5.4, where frequency of spawns is measured in spawns per thousand instructions. Having an ART size of one yeilds surprisingly good results. The frequency of spawns is decreased by only 7 percent when compared to infinite ART entries. An ART size of 2 comes very close, within 3 percent of the ideal case. Finally, allocating 8 entries to the ART actually has a very slight increase in the number of successful spawns. This effect will be discussed later.

Overall, this study has shown that by eliminating the ART stack, and using a new algorithm for prioritizing entries to be trained, the reconvergence predictor can be greatly simplified, use less power, and have its functionality extended so that it can track registers written across spawn points. This significant simplification can come at no cost to predictor accuracy, and very minimal cost in frequency of spawns for a two ART entry design.

Table 5.4: Spawn Frequency of New ART Prioritization Algorithm

	0	1	2	8
bzip2	5.471591	4.508201	5.554128	5.558232
gcc	34.321051	32.427426	33.977471	34.602616
crafty	31.702992	32.091354	31.922397	31.878318
eon	22.242634	19.230006	20.484265	22.194351
gap	34.772702	32.598785	33.907062	34.85344
gzip	19.621502	19.526233	19.603673	19.632441
mcf	33.144492	33.131231	33.087208	33.141089
perlbmk	31.476201	26.410081	28.763107	31.222603
twolf	29.211194	25.737746	28.395355	29.222949
vortex	25.874761	23.462467	25.365007	25.86044
vpr	19.778082	19.313379	19.716884	19.77755
average	26.14701836	24.40335536	25.52514155	26.17672991

CHAPTER 6

DYNAMIC VERSUS STATIC REVISITED

In Chapter 3, the difference between dynamic and static control independence analysis was compared briefly. The benefits of dynamic analysis were mentioned, but no examples were given for this particular predictor.

For the reconvergence predictor, there can be issues where an overly conservative spawn prediction gets trained for a branch, and it never gets cleared out of the table because it is actually the correct control independence point. A good motivating example is a normal if-then-else statement. If the "then" section of code has a condition check that would very rarely execute a return instruction, that would cause the true reconvergence point of the if-then-else to be the function return. While this may be the correct control independence point, common behavior of this code yields a control independence following the "then" and "else" sections of code. This point is the desired spawn point of the branch. However, once the return statement is executed, the reconvergence predictor will always predict a return as the spawn point, which is not useful.

In [9], they noted that their predictor's accuracy would increase if they only commited new trained entries to the predictor a certain factor of the time. This is caused by the effect mentioned in the previous paragraph, the predictor should only be trained on the very common behavior of the program, not the most infrequent behavior.

Another instance of this effect is measured when we limit the ART size as measured in Section 5.7.1. When we limited the ART size, we only train the reconvergence point of a branch a fraction of all instances of this branch. When the frequency of spawns is measured, it can be seen that the frequency actually improves when we are limited to eight entries in the ART instead of unlimited entries. This is another case where training on common behavior and skipping some of the infrequent behavior can cause an improvement in the predictor.

One study that was intended to enhance this effect was to clear the entire predictor table periodically. This was thought to remove infrequent behavior from the tables. There were some cases where this caused very slight improvements, but the penalty of losing all the good prediction data was too difficult to overcome.

More general ways at recording only common case behavior have been proposed, but the evaluation of these is outside the scope of this thesis. The first proposed way to capture this possible benefit is to keep two PC values for each type of reconvergence, the current value and the previous value. There will also be a confidence value for each of these PCs. If one of the values gets updated to a very conservative reconnection point, the previous value may still be a good spawn point a large percentage of the time. If its confidence value ever reaches a high enough value, we will then have the choice of choosing more common behavior over the most conservative behavior.

Another possible method to exploit common behavior over conservative behavior is to clear individual entries when they have been active a certain number of times. The method of clearing the entire table frequently was found to have some benefit, but the penalty of losing data on all branches was too great. Instead, we can clear the entries of a branch after every certain number of predictions. This will target clearing the entries for most common branches, where the penalty of retraining is small when amortized over all iterations. This method prevents us from clearing the entries of infrequent branches, eliminating most of the penalty seen when clearing the entire table.

CHAPTER 7

CONCLUSION

This thesis provided an in-depth study of the control independence prediction mechanisms necessary for a Polyflow processor. The requirements of a reconvergence predictor for this architecture require it to be able to predict control independence points on function calls, loops, and general control flow. This requirement is much more extended than the prediction algorithm in DMT, as a set of registers unsafe to rename on a spawn are also required.

The main contribution of this work is extending the algorithm proposed in [9]. This work proposed an algorithm for finding control independence points dynamically. Extentions to the algorithm were provided to target the specific needs of Polyflow, such as the unsafe register prediction, and reconnection distance prediction. In addition, modifications to the hardware design were made to significantly simplify the implementation of the predictor in hardware.

Some insight into the differences between dynamic and static information for control independence prediction was provided as well. Additionally, some future work has been laid out to explore the advantages of dynamic analysis more generally.

In total, it was found that this predictor finds an adequate frequence of spawn points to provide a Polyflow processor, with a very high accuracy level. Thus, the problem of finding control independence has been solved with a hardware only solution, showing how this particular piece of the architecture is easily implementable.

REFERENCES

[1] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996, pp. 191–202.

[2] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous mutithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392–403.

[3] H. Akkary and M. Driscoll, "A dynamic multithreading processor," in *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998, pp. 226–236.

[4] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative multithreaded processors," in *Proceedings of the 12th International Conference on Supercomputing*, 1998, pp. 77–84.

[5] P. Marcuello and A. Gonzalez, "Thread-spawning schemes for multithreaded architectures," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002, pp. 55–64.

[6] C. Cher and T. Vijaykumar, "Skipper: A microarchitecture for eploiting controlflow independence," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 4–15.

[7] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, 1992, pp. 46–57.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in program optimization," in *ACM Trans. Prog. Lang. Sys.*, 1987, pp. 319–349.

[9] J. D. Collins, D. M. Tullsen, and H. Wang, “Control flow optimization via dynamic reconvergence prediction,” in *Proceedings of the 37th Annual International Symposium on Microarchitecture*, 2004, pp. 129–140.

[10] D. Burger, T. Austin, and S. Bennett, “Evaluating future microprocessors: The simplescalar tool set,” University of Wisconsin - Madison, Tech. Rep. 1308, July 1996.