Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

2009

Dynamic Task Prediction for an SpMT Architecture Based on Control Independence

Komal Jothi

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Computer and Systems Architecture Commons](#), and the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Jothi, Komal, "Dynamic Task Prediction for an SpMT Architecture Based on Control Independence" (2009). *Dissertations and Theses*. Paper 1707.

<https://doi.org/10.15760/etd.1706>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

ABSTRACT

An abstract of the thesis of Komal Jothi for the Master of Science in Electrical and Computer Engineering presented February 19, 2009.

Title: Dynamic Task Prediction for an SpMT Architecture Based on Control Independence

Exploiting better performance from computer programs translates to finding more instructions to execute in parallel. Since most general purpose programs are written in an imperatively sequential manner, closely lying instructions are always data dependent, making the designer look far ahead into the program for parallelism. This necessitates wider superscalar processors with larger instruction windows. But superscalars suffer from three key limitations, their inability to scale, sequential fetch bottleneck and high branch misprediction penalty. Recent studies indicate that current superscalars have reached the end of the road and designers will have to look for newer ideas to build computer processors.

Speculative Multithreading (SpMT) is one of the most recent techniques to exploit parallelism from applications. Most SpMT architectures partition a sequential program into multiple threads (or tasks) that can be concurrently executed on multiple processing units. It is desirable that these tasks are sufficiently distant from each other so as to facilitate parallelism. It is also desirable that these tasks are control independent of each other so that execution of a future task is guaranteed

in case of local control flow misspeculations. Some task prediction mechanisms rely on the compiler requiring recompilation of programs. Current dynamic mechanisms either rely on program constructs like loop iterations and function and loop boundaries, resulting in unbalanced loads, or predict tasks which are too short to be of use in an SpMT architecture. This thesis is the first proposal of a predictor that dynamically predicts control independent tasks that are consistently wide apart, and executes them on a novel SpMT architecture.

THESIS APPROVAL

The abstract and thesis of Komal Jothi for the Master of Science in Electrical and Computer Engineering were presented February 19, 2009, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Douglas V. Hall, Chair
Dan Hammerstrom
Suresh Singh

DEPARTMENT APPROVAL:

Malgorzata Chrzanowska-Jeske, Chair
Department of Electrical and Computer
Engineering

DYNAMIC TASK PREDICTION FOR AN SPMT ARCHITECTURE BASED
ON CONTROL INDEPENDENCE

by
KOMAL JOTHI

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
2009

Contents

List of Tables	iv
List of Figures	v
List of Acronyms	vi
1 Introduction	1
1.1 Motivation	3
1.2 Thesis Contributions	8
2 Theory	10
2.1 Concept of Control Independence in Programs	10
2.2 Task Characteristics	14
3 Prior Work	17
3.1 Identifying Control Independence in Programs	17
3.1.1 Reverse Dominance Frontier	19
3.1.2 Control Independence in Superscalar Processors	21
3.1.3 Control Independence in Trace Processors	22
3.1.4 Dual ROB Implementation	24
3.1.5 Skipper	25
3.1.6 Exact Convergence	27
3.2 Thread Spawning Schemes in SpMT Architectures	28

3.2.1	Multiscalar Architecture	28
3.2.2	Superthreading	31
3.2.3	Trace Processors	34
3.2.4	Dynamic Multithreading	36
3.2.5	Speculative Multithreaded Processors / Clustered Speculative Multithreaded Processors	39
4	Task Predictor	45
4.1	Task Predictor Overview	45
4.1.1	Base Predictor	45
4.1.2	Enhancements on the Base Predictor	46
4.2	Task Predictor Implementation	48
4.2.1	Training Algorithm	51
4.2.2	Linking	54
4.2.3	Updating RPT from Multiple Cores	55
4.3	DOE Architecture	55
5	Simulation Methodology and Results	59
5.1	Simulation Environment	59
5.2	Task Spawning	60
5.3	Task Execution	64
5.4	Task Retirement	66
5.5	Task Squashing	70
5.6	DOE Experiments	70
5.6.1	Ideal Performance	70
5.6.2	Scalability	72

5.6.3	Performance for Various Levels of Parallelism	72
6	Conclusions	78
6.1	Comparison with Prior Work	78
6.2	Conclusions	81
6.3	Future Work	82
	References	85

List of Tables

5.1	Simulation setup	60
5.2	Distribution of task sizes	63
5.3	Impact of predictor accuracy on performance	66
5.4	Results for tracking convergence using three different methods . . .	67
5.5	Impact of wait time over predictor accuracy and performance	69

List of Figures

2.1	Types of control flow and reconvergence points in programs	11
3.1	An example of control independence	18
3.2	Control flow in a program	20
3.3	An example of exact convergence	27
3.4	Thread spawning in DMT	37
3.5	Thread ordering in DMT	38
4.1	Concept of linking reconvergence points	47
4.2	Task predictor block diagram	49
4.3	Reconvergence prediction table for the base predictor	50
4.4	DOE execution model	56
5.1	Linking to create a task of size greater than 1000 instructions	61
5.2	Speedup of 4-core DOE	71
5.3	Average core activity, 4-core DOE	72
5.4	DOE mcf speedup vs. number of cores	73
5.5	DOE twolf speedup vs. number of cores	73
5.6	4-core DOE twolf performance for various levels of parallelism	74
5.7	6-core DOE twolf performance for various levels of parallelism	75
5.8	8-core DOE twolf performance for various levels of parallelism	75
5.9	4-core DOE mcf performance for various levels of parallelism	76
5.10	6-core DOE mcf performance for various levels of parallelism	76
5.11	8-core DOE mcf performance for various levels of parallelism	77

List of Acronyms

AP	Above Potential
ARB	Address Resolution Buffer
ART	Active Reconvergence Table
BP	Below Potential
BTB	Branch Target Buffer
CD	Control Dependent
CFG	Control Flow Graph
CI	Control Independent
CIDD	Control Independent Data Dependent
CIDI	Control Independent Data Independent
CMP	Chip Multiprocessing
CSMT	Clustered Speculative Multithreaded Processors
DMT	Dynamic Multithreading
DOE	Disjoint Out-of-order Execution
FPC	Fork PC
ID	Identifier
JPC	Join PC

LPC	Link PC
LSQ	Load Store Queue
MSHR	Miss Status Holding Register
PC	Program Counter
PE	Processing Element
PHT	Pattern History Table
RAS	Return Address Stack
ROB	Reorder Buffer
RPC	Reconvergence PC
RP	Rebound Potential
RPT	Reconvergence Prediction Table
SM	Speculative Multithreaded Processors
SMT	Simultaneous Multithreading
SpMT	Speculative Multithreading
TLP	Thread Level Parallelism
TPC	Targeted PC
TU	Thread Unit

Chapter 1

Introduction

A computer uses a microprocessor to run software programs written for a wide range of application areas. There is a growing demand for these applications to perform better, which if achieved, will increase their utility, and make available certain applications which were considered impractical in the past. An example of such an application is video conferencing between two parties at different locations. Better performance of these programs translates to executing them faster on processors. One way to make these programs run faster, is to make use of the advancements in semiconductor device technologies to build faster switching circuits, thereby reducing the time needed to execute the program. Another solution is to make use of the shrinking device technologies to provide more computing resources on the same processor, thereby making programs run faster. However, modern microprocessors employ more advanced architectural techniques to provide a solution. This solution is based on an analysis made on the nature of program behavior. A study of program behavior shows that application programs, for most part, do not behave randomly, but instead, exhibit regular patterns of behavior. Two concepts that make use of this regularity are caching and branch prediction.

It is seen that programs exhibit the property of temporal locality (the same memory location being accessed repeatedly within a short time) and spatial locality (the neighbouring memory locations being accessed within a short time). A cache exploits this property by placing a small memory with short latency with recently

accessed memory locations, for faster access and better performance.

Another example of exploiting regularity in programs is the concept of branch prediction. Normally, in a pipeline, if a branch instruction is encountered, control logic will have to wait until the direction of the branch is resolved, before it tries to fetch the next instruction. During this time, the pipeline remains stalled, with none of the instructions moving ahead in execution. With branch instructions being very frequent in general purpose programs, this stall severely affects performance. But, because of the fact that branches exhibit regularity most times, the outcome of a branch can be confidently predicted beforehand, based on its behavior on previous occasions. As a consequence of this prediction, instructions can be fetched from the predicted direction before computing the branch outcome, without stalling the pipeline, and thereby allowing faster execution.

Both these techniques have worked very efficiently on all major processors, hence showing the potential of prediction to improve the performance of application programs.

In general, it is useful to observe program behavior, to exploit a behavioral pattern to improve performance. There are other techniques like data dependence prediction, data value prediction, load/store address prediction and memory disambiguation prediction that take advantage of regularity in programs. In this thesis, we propose a dynamic task predictor that will partition a single sequential program into multiple tasks. The task predictor will predict the next likely task that will follow the currently executing task. These tasks are concurrently executed on an architecture following a novel execution model.

The following section presents an overview of some of the existing architectures, their shortcomings, and the motivation for our work.

1.1 Motivation

An in-order pipelined processor suffers from two major limitations - 1.) Stalling due to change in control flow. 2.) Stalling due to long latency operations.

The implication of the change in control flow is that it would waste processor cycles to find the outcome of the branch instruction. One way to overcome the control flow limitation is branch prediction. It was determined that predicting the direction of control flow and speculatively executing instructions would give more performance rather than waiting for the branch outcome to be resolved. The key rule that was followed was to make the common case execute faster.

As a next step, superscalar processors were introduced to increase performance, by building wider machines with more pipeline resources that could fetch, issue and execute more instructions in a single cycle. Sequential superscalar processors completely relied on the compiler to generate an optimal instruction schedule to allow multiple independent instructions to be issued every cycle. But sequential processors still had to deal with the problem of long latency operations stalling the pipeline and affecting performance negatively. A short latency instruction or an instruction with ready inputs behind a long latency instruction in the pipeline would stall, even if execution units were available, until the long latency instruction would move further ahead.

Out-of-order execution was a technique that was introduced to deal with the

problem of long latency operations stalling the pipeline. In order to achieve parallelism, out-of-order superscalars maintain a large associative buffer, called the instruction window. Instructions from this window are issued in a non-blocking data flow manner, thereby allowing better utilization of resources. This helps out-of-order processors to achieve a performance higher than in-order processors. In order to schedule instructions from this instruction window, out-of-order processors use different algorithms, mainly Scoreboarding [1], Tomasulo's algorithm [2] and Register Update Unit (RUU) [3]. A reorder buffer (ROB) is used to reorder the instructions back in program order and perform exception recovery. Out-of-order processors also use the concept of register renaming to reduce the effect of false dependencies in programs, namely WAR and WAW hazards, and overcome the limitation that sequential processors face due to the shortage of logical registers, at the same time requiring no change in the ISA. One problem with out-of-order processing is the design complexity, which reduces the maximum processor clock rate. In comparison, in-order processors have a very simple design, hence they can be clocked at a higher frequency. Out-of-order processors needed to ensure that the decrease in the clock rate was offset by the increase in performance.

Performance of superscalar processors suffers because of three main reasons,

- *1. Inability to scale:*

It is a very centralized architectural approach. As a consequence, is not possible to scale these processors. When the size of these structures is increased in order to improve performance, there is a direct impact on the critical path of the processor pipeline. We try to explain the reason in more detail now. The register file in a typical processor has 2 read ports and 1 write port

for every instruction being issued in a cycle (assuming that an instruction has 2 source operands and 1 write operand). Thus making the issue width n implies that the register file has $2n$ read ports and n write ports. The interconnection path from the execution units to the register file, as also the bypass network falls in the critical path of the processor, and decides the cycle time. Getting into a little more detail, the reason why this interconnect is a problem is because, as we move to newer technologies, though transistor sizes decrease, the onchip interconnects do not scale accordingly. This is because, while the transistor performance is not affected due to their shrinking size, thinning the metal interconnects decreases the speed at which the signal can propagate along the wires. This is because the resistance offered by the wires increases when the width of the wire decreases [4].

- *2. Sequential fetch mechanism:*

The technique to maintain an instruction window to overcome the problem of long latency instructions blocking pipeline resources, has its own limitations. This is because of the imperative way in which general purpose applications are written. Instructions that lie close to each other will invariably be dependent on each other. For more parallelism designers will need to look very far ahead from the instruction that created the stall. A superscalar, with its sequential fetching mechanism, combined with the inability of its instruction window to scale, is seriously limited in its capacity to look for parallelism among instructions.

- *3. High branch misprediction penalty:*

In a superscalar, more the instructions fetched, higher will be the probability

of finding more independent instructions to execute in the same cycle. To fetch more instructions, branch prediction is used to overcome the limitations imposed by the change in the direction of control flow. To fetch correct instructions, the processor would need a very accurate branch prediction mechanism that would predict several branches even before the first one is resolved. This calls for a lot of hardware budget on the branch predictor. The problem with this approach is that even with all this there are still branch mispredictions. With faster designs, deeper pipelines, and more instructions being executed each cycle, the branch misprediction penalty keeps getting higher. There are two events that follow a branch misprediction - 1.) All the instructions in the pipeline that were fetched after the branch are flushed. 2.) Control logic looks for instructions from the correct path and inserts them into the pipeline. While all this happens, the pipeline remains stalled without doing any useful work, which impacts performance.

Two approaches were suggested to deal with two different limitations of superscalar processors. Conventional forms of both these architectures used the concept of processing instructions from multiple applications in the same cycle to overcome the sequential fetch limitation of superscalars.

Simultaneous Multithreading (SMT) [5] was suggested to deal with the problem of latency. SMT architecture executes instructions from multiple applications in the same cycle, each application executing on a separate thread. All the threads share centralized resources. If one thread encounters a long latency instruction, then another thread, if available, uses the issue and execution resources. There is no delay involved in context switching because all the contexts or threads are active at the same time. This way there are no resources that are stalled for a

long time. However the same problem of resource centralization still persists with different threads sharing the same register file, issue slots and execution units. Since this has a direct impact on the clock cycle, SMT architecture always runs on a slower clock. Scaling SMT architecture is also a problem.

Chip Multiprocessing (CMP) [6] was another architecture that was suggested to overcome the scaling limitation in superscalars. The idea was to decentralize the architecture to support multiple threads on multiple simple cores. This architecture has the advantage of being simple, requiring lesser engineering effort. The architecture could be run with a faster clock, and could scale well because of decentralized resources solving the interconnection problem. However, this architecture still has to deal with the problem of latency. CMP, being a very fixed assignment architecture, processing units are allocated exclusively to a thread. As a result pipeline resources will idle in case of a hazard, leading to their under-utilization.

One major disadvantage of both SMT and CMP is that they work well only on parallel applications and not on sequential applications. But, sequential programs constitute a major portion of existing and legacy applications (ease of programming and portability being the reasons) that run on microprocessors, and it is not feasible to rewrite these programs for a different architecture. Parallelizing compilers used to generate code that can run on these architectures are not very efficient. This called for an architecture that could provide high performance on sequential programs.

Speculative Multithreading (SpMT), pioneered by the Multiscalar architecture [7], provided a solution to this problem of finding sequential programs that could

run on a decentralized architecture. The Multiscalar architecture follows a decentralized execution model, partitioning a single program into multiple tasks and executing these tasks on separate processing elements concurrently. However, the Multiscalar followed a compiler based scheme to create tasks. Consequently, a major disadvantage was the need to recompile the application binaries in order to run them on this architecture.

It is desirable to have a dynamic scheme that could partition a single program into threads at run time without support from the compiler (making the architecture independent of the compiler). In this regard, we study several architectures like Dynamic Multithreading (DMT) [8], Trace processors [9], Speculative Multithreaded Processors (SM) [10] and Clustered Speculative Multithreaded Processors (CSMT) [11], that were proposed to meet this requirement. However, the dynamic schemes proposed so far depend on program constructs like iterations of loops, end of loops and procedures to predict threads that can run concurrently on multiple processing elements, be it centralized (DMT) or decentralized (SM, CSMT) architectures. Studies have shown that there is a lot of parallelism in programs independent of program constructs. Lam et al. [12] showed that following control independence and multiple flow can provide considerable amount of parallelism. Based on these studies, the main focus of this thesis is dynamic partitioning of sequential programs on the basis of control independence. The contributions of this thesis are outlined in the following section.

1.2 Thesis Contributions

- *Dynamic task prediction for SpMT architecture based on control independence:*

We propose a hardware predictor to identify control independent points to spawn tasks. Our predictor is based on a design proposed in [13], extended to predict task size and to link multiple control independent regions to form larger tasks. The predictor is also extended to work with an SpMT architecture, with capability to update its tables from multiple cores.

- *Simulation model that predicts performance under various levels of task parallelism:*

The tasks from our predictor are executed on a Disjoint Out-of-order Execution (DOE) architecture [14]. DOE is a novel implementation of SpMT with each thread unit being a simple in-order core with latency tolerance [15] and checkpoint processing [16] features. The DOE model is explained in more detail in section 4.3. We exploit the simplicity of the core, the separation of task execution into completely disjoint independent and dependent data threads, and the decoupling in DOE of performance and data communication timing to measure DOE performance for various hypothetical levels of parallelism. This gives valuable insight into the amount of task parallelism required for good performance and can help guide the design optimization and task selection process in a real hardware implementation of DOE.

The rest of the document is organized as follows. Chapter 2 details the theory behind control independence and factors that influence task creation for an SpMT architecture. Chapter 3 details the related work leading up to our work. Chapter 4 details our task predictor. Chapter 5 details our simulation methodology, experiments and results. Chapter 6 compares our work with prior work, details the potential for future work and concludes.

Chapter 2

Theory

This chapter begins with a detailed explanation of the concept of control independence in programs. This is followed by a note on the factors that need to be considered while spawning threads on any SpMT architecture.

2.1 Concept of Control Independence in Programs

Since our task predictor is based on control independence in programs, let us look at this concept in detail. Control independent point for a branch in a program is a future dynamic instruction that is reached eventually irrespective of any intervening control flow. There are different types of control flow with corresponding reconvergence points as shown in Figure 2.1. They are explained below.

- 1.) *Exact convergence:* These are simple if-then program constructs. Branch B1 is an example of such a program construct which reconverges at point R1.
- 2.) *Diamond or hammock structures:* These are if-then-else program constructs. Branch B2 is an example of such a program construct which reconverges at point R2.
- 3.) *Complex control flow:* In Figure 2.1, B4 is an example of a branch that exhibits complex control flow. It is interesting to notice the reconvergence point for branch B5. It is generally seen that the reconvergence points are below the

Figure 2.1: Types of control flow and reconvergence points in programs

targeted branch because application programs are usually structured as a top to bottom flow of program constructs. But sometimes the reconvergence point can be above the targeted branch. It is also interesting to notice the case where control flow leads to a point below the reconvergence point R4 and then rebounds back to R4.

4.) *Procedure returns*: A branch inside a procedure can hit a return instruction before it reaches its reconvergence point. In such a case the instruction immediately following the procedure call invocation will be the reconvergence point. Consider the fact that a procedure can be called from different parts of the program. Hence the reconvergence point for a branch that encounters a return instruction will depend on what part of the program invoked the call to the procedure. Hence we need to track call levels when we train branches for their reconvergence points.

From the above discussion and an analysis performed by means of a survey of control flow in programs by Collins et al. [13], it can be established that there are four categories of potential reconvergence points for a branch condition. Any branch in a program can have its reconvergence point included in one (or more) of the following categories. For the rest of the document, PC (meaning program counter) refers to the address at which the program instruction is located. The term "below" refers to an instruction that is located at a higher PC than the reference instruction. Similarly "above" refers to a lower PC than the reference instruction.

We introduce some terms in relation to control independence [13].

rec below branch indicates that the reconvergence point lies below the targeted branch.

rec below max refers to that PC, below which no instruction can execute after the targeted branch executes, before executing the PC at *rec below max*. In other words, it is guaranteed that once the targeted branch executes, irrespective of the control flow, *rec below max* will definitely execute before any other PC below *rec below max* will execute.

rec above branch indicates that the reconvergence PC lies above the targeted branch.

rec above max refers to that PC, below which no instruction can execute after the targeted branch executes, before the *rec above max* PC executes. The only difference between *rec above max* and *rec below max* is that the reconvergence point is located above the branch in the *above* case and below the branch in the *below* case.

rebound reconvergence is a subset of *rec below branch*. Sometimes control flow will branch over the reconvergence PC and later on rebound backwards to the reconvergence PC (as seen in branch B4 in Figure 2.1). In some cases, control flow will directly lead from the targeted branch to the reconvergence point (as shown by the control flow arc C4 in Figure 2.1). *rebound reconvergence* refers to such a reconvergence PC.

return reconvergence refers to the reconvergence PC of a return instruction in

a procedure. A branch inside a procedure can hit a return instruction before it reaches its reconvergence point. In such a case the instruction immediately following the procedure call invocation will be the reconvergence PC.

2.2 Task Characteristics

In the introduction, we mentioned that an SpMT architecture partitions a single program into multiple blocks of instructions to be executed concurrently on separate processing units. Let us call each block of instructions a *task*. A task is a continuous block of instructions in the dynamic instruction stream that has one entry point which is the first instruction of the task. Task partitioning is an important step because it could increase performance significantly if there is enough parallelism to take advantage of the available cores. The fundamental characteristics of a task are its size, control independence and data independence from its previous task. This section discusses the problems that could occur on a multiprocessor architecture and how they are related to task characteristics [17].

Task start overhead / Task commit overhead: Task start overhead is the time involved in probing a potential spawn point in the task predictor, finding its reconvergence point, and communicating the live in registers. In addition to this, hardware parameters of the core (latency of the pipeline resources) also influence task start overhead. Task commit overhead is the time involved in committing the large speculative state to the architectural state.

Impact of task size: If the task size is very small, task start overhead will occupy a significant portion of execution time, and will bring down the performance benefits

of multiple flow. If the task size is very small, it may be difficult to find enough parallelism within such a short distance.

A large task size implies sufficient distance and parallelism between two tasks. However, if the task size is very large, then the misspeculation penalty is higher, because more instructions are thrown away. Also, if the task size is large, the amount of branch mispredictions in the dependent thread and the amount of data misspeculations would be correspondingly large. Either of these two situations leads to squashing the task. This is a severe penalty. Also a large task represents lost opportunity, because it could have been instead executed concurrently on parallel cores. This however may not be true if most of the tasks are of approximately equal size, which brings us to the point of load balancing.

Load imbalance: A small task following a large task will have to wait for a long time (without doing any active work) until the large task commits all its instructions, because the tasks have to be retired in program order to maintain overall sequential execution. It is important to minimize this stall time by keeping all the cores active with balanced loads.

Control flow misspeculation: If the tasks are partitioned along control independent boundaries, then local branch mispredictions will not affect other tasks. However, the penalty is heavier on a task misprediction, because in a multiprocessor architecture, more instructions would have been executed speculatively by multiple cores. It is also a good idea to use easy to predict reconvergence points as task boundaries, because unlike branch prediction, task prediction does not require all branches to be predicted.

Inter task data dependence latency: In case of an inter task data dependence, this is the time involved in computing the result for the producer instruction.

Inter task data communication latency: In case of an inter task data dependence, this is the time involved in communicating the result of the producer instruction.

In any SpMT architecture, all these factors need to be considered by the mechanism (either compiler or hardware) that partitions a program into tasks.

Chapter 3

Prior Work

The task predictor proposed in this thesis partitions a sequential program into tasks based on control independence. In the first part of this chapter, we will review the prior work that has been done to exploit control independence for better performance. In the second part of this chapter, we will look at the thread spawning schemes followed by different SpMT architectures that focus on increasing the performance of single thread applications.

3.1 Identifying Control Independence in Programs

Lam et al. [12] performed a limit study on parallelism in sequential programs, setting an upper limit on performance with unlimited resources. They concluded that limitations on parallelism set by control flow can be overcome with three different concepts - 1.) Speculative execution 2.) Multiple flow 3.) Control independence.

Speculative execution has been employed by almost all modern processors including out-of-order superscalars. Multiple flow has been exploited by architectures like SMT, CMP and SpMT. We will discuss more about multiple flow in section 2.2. In this section, we will see how the concept of control independence in programs has been used in different implementations.

Most of the work reported in studies have used the concept of control independence to reduce branch misprediction penalty. Let us look at how it was done.

Figure 3.1: An example of control independence

Consider the program execution shown in Figure 3.1. Basic blocks 2 and 3 converge at basic block 4, or equivalently control flow in basic block 1 reconverges at 4. Instructions within basic block 4 are independent of control flow, and hence called control independent (CI) instructions. Instructions within basic blocks 2 and 3 are dependent on the control flow, and hence called control dependent (CD) instructions. Instructions in the CI region which are data dependent on instructions from the CD region are called control independent data dependent (CIDD) instructions. Instructions in the CI region which are data independent of instructions from the CD region are called control independent data independent (CIDI) instructions. In a conventional out-of-order superscalar implementation all the instructions following the mispredicted branch are squashed from the pipeline, regardless of whether they are from the CD or CI regions, which would mean that the CIDI instructions will be needlessly processed again.

Most of the implementations discussed in this section were proposed to reduce

the penalty caused by a mispredicted branch by selectively executing only the CD and CIDD instructions. We will not discuss on how these proposals reduce branch misprediction penalty in detail. Our main focus is to study the algorithm followed by these proposals to detect the control independent point for a branch instruction. We will start with the well know compiler technique, the Reverse Dominance Frontier algorithm [18]. Section 3.1.1 explains the scheme used in [12] to detect control independence.

3.1.1 Reverse Dominance Frontier

This algorithm determines the most distant control dependent point for a branch instruction. This will give the first control independent instruction of that branch instruction. Control dependence detection in [12] is performed in two stages; control dependence detection within each procedure and inter procedural control dependence detection. The Control Flow Graph (CFG) is constructed from the program binary. From the CFG, basic blocks, basic block boundaries and successors of each basic block are identified. It is clear that all the instructions within a basic block will be control dependent on the branches that lead into the basic block. An instruction may be control dependent on many branches if the overall CFG is considered. However, each dynamic instance of an instruction depends immediately on only one of these branches. For example, instruction 2 in Figure 3.2 is control dependent upon both instructions 1 and 5. If control flows from instruction 1 to instruction 2, only the dependence on instruction 1 needs to be considered. This is accomplished by sequentially numbering each basic block in the CFG, analyzing the CFG and recording for each basic block the sequence number of its most recent instance. The immediate control dependence of an instance of an instruction is

Figure 3.2: Control flow in a program

simply the branch with the latest sequence number leading up to the instruction. This branch with the latest sequence number is said to be the branch within the reverse dominance frontier of the instruction.

Interprocedural control dependences are handled by maintaining a stack that contains the control dependence information for each active procedure. This stack records the control dependence for each calling instruction and the sequence number at the start of each procedure. Each procedure inherits the control dependence of the instruction that calls that procedure. Without recursion, the control dependence for an instance of an instruction is either the control dependence on the top of the stack or an instance of a branch within its reverse dominance frontier, whichever is most recent.

With recursion, the control dependence for an instruction is either the dependence on the top of the stack or an instance of a branch within its reverse dominance frontier from the same procedure invocation.

3.1.2 Control Independence in Superscalar Processors

Since the work done in [12] was a limit study, practical implementation issues were not considered. The entire dynamic instruction stream was scheduled at once, assuming unlimited fetch, issue and execution resources. Rotenberg et al. [19] studied the importance of control independence in superscalar processors with a practical implementation of a processor. The impact of branch mispredictions was separated into two parts - 1.) Penalty paid in the form of wasted resources on instructions from the wrong path. 2.) Penalty paid due to the data dependencies on instructions from the wrong path.

It was concluded that both these components play a major role in reducing the performance of superscalar processors, the penalty due to resource wastage being significantly more dominant of the two components.

Their proposal to reduce branch misprediction penalty is as follows. A CI point for a mispredicted branch is detected with the Reverse Dominance Frontier algorithm. After separating the CD and CI instructions, CD instructions are inserted/removed from the middle of the ROB. This is achieved with a linked list implementation of the ROB. It was suggested that if a single instruction linked list ROB makes the implementation complex, then the ROB could be partitioned into multi-instruction slots. Following this, only the CIDD instructions are selectively reissued. To reduce the complexity in reissuing CIDD instructions, CIDD instructions remain in the issue buffers until they retire, and reissue if there is a change

in their source values.

They also proposed a solution to find CI points using only hardware. Program constructs like loop and function boundaries were used as global reconvergence points. It was pointed out that, although these are not precise or the nearest reconvergence points of the targeted branches, they are easily detectable with simple hardware.

3.1.3 Control Independence in Trace Processors

Rotenberg et al. [20] continued with their research to use control independence in Trace Processors with an objective to reduce branch misprediction penalty. Considering the Trace processor was a multiple flow architecture, it is interesting to note that the concept of control independence was not used to spawn parallel threads.

The trace processor execution model is explained in section 3.2.3. On a trace cache hit, the trace is assigned to a processing element (PE). If a branch mispredicts, the trace predictor is moved back to that trace, and the trace buffers starts repairing that trace. Trace predictors follow a very similar strategy to that discussed in section 3.1.2 for misprediction recovery. The only difference is in the organization of the ROB. To repair a trace, they perform arbitrary insertion/removal of instructions from the middle of the window by ordering the PEs in the form of a linked list.

After a misprediction is repaired, execution is resumed in one of the two following ways depending on whether the CI point was covered in a fine grain or coarse grain manner.

1.) *Fine Grained Control Independence (FGCI)* -

This is a type of control independence that will reconverge within the same trace. In this case, recovery is simple because the PE arrangement is unaffected. Control logic dispatches the repaired trace to the affected PE.

Small if-then, if-then-else, and nested if-then-else constructs that do not contain loops or function calls are traces that satisfy FGCI. This is because, firstly, they have fixed-length and relatively short control dependent paths, most of which fit within a trace. Secondly, these regions can be precisely and efficiently detected by hardware because of the regular patterns they exhibit.

They identify reconvergence points only for forward branches with a hardware algorithm. Whenever a forward branch is encountered, the most distant taken target until then (recognized by the PC value) is identified as the reconvergence point. The path length from the branch to the reconvergence point is measured by profiling and the maximum length is stored. Trace selection then uses this information to pad any selected path until its length matches the longest path. By equalizing path lengths, trace selection synchronizes control dependent paths at the reconvergent point. The branch is not a candidate for FGCI if the path length exceeds the maximum trace length before reconvergence, or if a backward branch, function call, or indirect branch is encountered before reconvergence. If the branch is not a candidate for FGCI, then it has to be tracked with CGCI.

2.) *Coarse Grained Control Independence (CGCI)* -

This is a type of control independence that will extend beyond a single trace. To deal with this condition, easily detectable loop and function call boundaries are

treated as globally visible reconvergence points. To ensure trace level reconvergence for CGCI branches, traces are made to start at chosen global reconvergent points. So the traces between the mispredicted branch and the globally 'exposed' reconvergence point are squashed and the PEs are deallocated. Correct CD traces are allocated to the freed PEs. If the freed PEs are not sufficient, then PEs are reclaimed from the tail of the window.

3.1.4 Dual ROB Implementation

In a dynamic control independence detection scheme Chou et al. [21] proposed a dual ROB implementation to save on the CIDI instructions from being fetched into the pipeline after detecting a mispredicted branch.

The implementation of the idea is as follows. A duplicate ROB called the dynamic control independence buffer (DCIB) is maintained, which keeps track of each instructions ROB tag, destination register and physical register mapping. The DCIB has three pointers namely, head pointer, mispredicted branch (MP) pointer and the control independent instruction (CI) pointer. When a branch mispredicts, control logic rolls back execution to the mispredicted branch in the ROB. In the DCIB, the MP pointer is moved to this branch, so that the CI instructions between the MP pointer and the head pointer can be detected. The fetched instructions are compared with the instructions in the DCIB. When there is match the algorithm assumes that the CI point is reached. Now the CI pointer is made to point to the next instruction of the matching instruction. On each successive match, the CI pointer is incremented to point to the next instruction. This makes the search process easier. Subsequent instructions do not have to associatively search the DCIB for a matching instruction. They only have to compare themselves with the

instruction pointed by the CI pointer. If the CI pointer does not match the fetched instruction, the MP pointer is made to point to the last matching instruction to enable detecting non-contiguous CI instructions.

It was concluded that this implementation can give better performance due to two reasons - 1.) It saves on the execution resources of CIDI instructions. 2.) These instructions feed their results to their consumer instructions earlier.

Since this was one of the earliest implementations to detect CI points, it is a very basic idea without any heuristics. It is not efficient to duplicate the ROB resources. It is not efficient and fast to do an associative search of the DCIB to find a match with the fetched instruction. Reduction in penalty is only on resources spent on CIDI instructions, which could result in a very small performance improvement.

3.1.5 Skipper

The study in [19] concludes that in the event of a branch misprediction, the biggest performance limiter is wasted resources consumed by incorrect control dependent instructions. To conserve these resources, Cher et al. [22] proposed an implementation that avoids incorrect instructions by skipping over, without even fetching instructions from the CD region of a difficult to predict branch. Instructions are fetched from the CI point, which will be executed irrespective of the outcome of the branch. CD instructions are executed only after the difficult branch is resolved. Execution of the CIDD instructions is delayed until the difficult branch is resolved and CD instructions are executed.

A difficult to predict branch is identified with saturating counters, which indicate the confidence with which the branch can be predicted. The reconvergence point for this branch is detected using heuristics based on control flow patterns

generated by compilers for conditional branches (explained further). Even though instructions are fetched out-of-order, program order is maintained by creating an appropriate gap in the ROB and load store queue (LSQ), to be filled later by the skipped instructions. To detect CIDD instructions, in order to make them wait until the CD instructions execute, data dependences are estimated by learning from earlier dynamic instances. Profiling is also used to generate the influenced and live-in register masks from the skipped instructions.

The heuristic followed by the compiler is as follows. For if-then-else type of branches, the compiler generates a branch (let us call it BX) to determine whether the if block or the else block should be executed. The compiler also generates a jump to the reconvergence PC (RPC), and places this jump instruction at the end of the if block, so as to elide the instructions in the else block. Hence the RPC can be determined if this jump is located. The target of the branch BX is the else block and the jump instruction is located immediately before the target. The taken target of this jump instruction is the RPC.

If the instruction immediately before the branch BX is not a jump statement, then it is assumed that the difficult to predict branch is of the if-then construct. For if-then constructs, the compiler generates a jump instruction to elide the if-clause instructions that would be executed if the if condition is false. In this case the target of this jump instruction is the RPC.

If a difficult to predict branch is a backward branch, the RPC is the instruction immediately following the backward branch.

Figure 3.3: An example of exact convergence

3.1.6 Exact Convergence

A subset of control independence called exact convergence was used by Gandhi et al. [23] to reduce branch misprediction penalty. A set of branches exhibit the behavior shown in Figure 3.3. In such branches, the CI point coincides with the start PC of the corrected CD path. Consequently, if such branches mispredict, it is not necessary to nullify the data dependences from the incorrect CD path and re-establish true data dependences from the correct CD path. Their study indicates that such branches occur frequently (32%) in programs.

Their algorithm to detect the CI point for a branch that exhibits exact convergence is as follows. Each branch is stored in an Alternate Target Buffer (ATB) in addition to the Branch Target Buffer (BTB). ATB stores the alternate target of a branch, i.e. the taken PC if the branch is predicted to be not-taken and the next sequential PC if the branch is predicted to be taken. The address of every fetched instruction is searched in the ATB. On a hit, the fetched instruction is a potential exact convergence point. If the branch corresponding to the ATB entry that hit is

found to have mispredicted, the recovery sequence is started.

Collins et al. [13] proposed a general technique to dynamically identify reconvergence points which could be applied to any performance optimization scheme. We use their technique as our base and propose extensions in section 4.1.2. Transparent Control Independence is a recent proposal from Al-zawawi et al. [24] to reduce branch misprediction penalty using control independence. They use the same technique as our base predictor to detect CI points. Multiscalar architecture does not have any explicit heuristics to detect CI points [17].

3.2 Thread Spawning Schemes in SpMT Architectures

In this section, we will discuss about architectures that followed multiple flow, by partitioning a single sequential program into multiple threads. We are mainly interested in studying the thread spawning scheme followed by each architecture. We will also take a brief look at the execution model of each architecture, which will give us some insight into understanding the motivation behind their thread spawning scheme.

3.2.1 Multiscalar Architecture

Execution model

The concept of Multiscalar architecture [7] is to connect multiple processors in a decentralized manner, to achieve overall multiple issue from a large decentralized instruction window. The absence of centralized structures removes the need for wide associative searches and wide data paths. The cores are organized in the

form of a ring. A single program is partitioned into multiple tasks by the compiler. Tasks are spawned in sequential order. Tasks are retired in program order to maintain sequential semantics. If a task is mispredicted, the incorrect task and all subsequent tasks are squashed. The register dependencies are easily identified with the compiler and routed dynamically around the ring with compiler generated masks. The memory dependences however are not easily identified with the compiler. Even then, loads are issued aggressively for better performance. An address resolution buffer (ARB) is maintained to keep track of loads and stores being performed in all the processors. In the event of a misspeculation, the violating task and all subsequent tasks are squashed.

Thread spawning scheme

The compiler partitioning scheme is explained below [17]. The starting process for this scheme is traversing the CFG starting from its root. Then heuristics are added to this basic process to create better tasks. These heuristics can be classified into three categories - task size heuristics, control flow heuristics and data flow heuristics.

Task size heuristic:

The first step in the scheme is to start a task with a single basic block. Multiple basic blocks are included to increase the task size. A threshold (both maximum and minimum) is maintained to limit the task size. The merits and demerits of large and small task sizes were discussed in section 2.2. Loops bodies that are smaller than the threshold are unrolled, and calls to function with bodies smaller than the threshold are inlined in the same task. Otherwise entry into loops, exit

out of loops, call to functions and return from functions terminate tasks. When more basic blocks are added into the same task, it results in inter task control flow misspeculations (the reason is explained further). To reduce control flow misspeculations, the number of successors of a task are controlled with the control flow heuristics.

Control flow heuristic:

The successors of a basic block are examined to see if they can potentially be included in the same task to increase its size. The maximum number of successors for a task is set as a threshold. The maximum successors from a task is limited by the size of the table used to perform task prediction. If the number of successors are more than what the table can manage, accuracy of prediction decreases. It is desirable to have a good number of basic blocks, without increasing the number of task successors. Basic blocks are added (following a greedy algorithm) even if the count of successors goes beyond the threshold, anticipating a reconvergent basic block, that will eventually reduce the number of successors to fall below the threshold. The final number of successors are determined when all the control flow paths terminate. At this point a feasible task is one with a good size and minimal number of successors.

Data flow heuristic:

Control flow heuristic includes basic blocks if the number of successors remains within the threshold, without considering their data dependencies. The goal of data flow heuristic is to include a basic block in a task if that basic block includes a consumer instruction, whose producer is already included from another basic

block in the same task. If the producer and consumer are not in adjacent basic blocks, then it tries to add all the basic blocks in between to the same task. If the producer and consumer are in different basic blocks and it is not possible to include them within the same task, the heuristic tries to control the scheduling of instructions such that, the producer is scheduled early and the consumer is scheduled late during execution in their respective tasks.

It is possible that each of the heuristics have contradicting requirements. To solve this problem, profiling is used to integrate the rules.

3.2.2 Superthreading

The Multiscalar architecture suffers from two key issues - 1.) The ARB is a large and complex structure to implement. 2.) The penalty on a memory dependence misspeculation is large, since the violating task and all subsequent tasks are squashed.

The Superthreaded architecture [25] focussed on reducing this penalty by not speculating on data dependences and hence reducing hardware complexity. Instead, it enforces data dependences and speculates only on the control dependences. The details of the architecture are given below.

Execution model

The Superthreaded architecture is a decentralized architecture, with each thread processing unit having its own program counter, issue and execution resources. The compiler statically partitions the CFG of a program into threads along loop iterations. The non speculative thread forks speculative threads, which in turn can

fork further speculative threads. Threads retire in original sequential program order. The key feature of this architecture is that it uses a pipelined thread model to enforce data dependences between concurrent threads. The execution of a thread is partitioned into four main stages which are briefly explained below.

Continuation Stage: After a thread is initiated by its previous thread, it starts with the continuation stage. The function of the continuation stage is to compute induction variables needed to fork the next thread. This is followed by the fork instruction that initiates the next thread with the results from the continuation stage.

Target Store Address Generation (TSAG) Stage: This stage computes the addresses of store operations in the current thread, on which subsequent threads may be data dependent. These store addresses are stored in the memory buffer of the current thread and subsequent threads. After all the store addresses are computed and communicated to subsequent threads, the current thread issues a synchronization instruction, which indicates the subsequent threads to start computations dependent on previous threads. Until then the subsequent threads can only perform computations that are not dependent on previous threads (except those values communicated after the continuation stage). Explicit compiler instructions are used to make sure instructions without data sources do not execute.

For better performance, the TSAG stage is divided into two parts. The first part is to generate store addresses that do not have any data dependences on previous threads, and hence can be computed and communicated faster. The second part is to generate store addresses that are dependent on the previous thread, which

have to wait for the synchronization instruction from the previous thread.

The compiler performs code motion within a thread to promote instructions that compute store addresses to the TSAG stage. For threads that may have many control flow paths, the compiler also moves the condition tests of branch instructions to the TSAG stage, so that only the required store addresses to be used in the chosen path are computed. The results from these computations are later used in the Computation Stage, explained next.

Computation Stage: This stage performs all the remaining computations of a thread. These computations are forwarded to the memory buffer of the current and subsequent threads. Data forwarding between threads is done with communication instructions inserted by the compiler.

Writeback Stage: In this stage, if the control speculation is found to be correct, the thread is retired. Its state in the memory buffer is committed to the cache. To maintain correct memory state, concurrent threads commit the state of their memory buffer to the cache in program order. Explicit synchronization instructions are issued by the compiler to ensure this order. The compiler also inserts instructions to squash a thread execution in case of a misspeculation, or terminate a thread normally in case there are no exceptions.

Thread spawning scheme

The compiler examines the CFG to identify potential candidates to spawn threads. A good candidate is a group of instructions that perform a round of computations on a set of data. If there is considerable amount of computation to be done after

the computation and TSAG stages, the compiler will initiate a new thread. This architecture spawns only on loop iterations.

3.2.3 Trace Processors

Execution model

The Trace processor [9] is a decentralized architecture with each processing element (PE) having its own fetch, issue and execution resources, register file and ROB. The execution model of the processor is organized on the basis of traces. A sequential program is partitioned into traces and stored in a trace cache [26]. A trace is a sequence of instructions observed in the dynamic instruction stream that can include different basic blocks which are far apart in the static representation of the code. A trace can contain any number of branch instructions. Different paths taken by the same set of branches are stored as different traces in the trace cache. The same basic block may be redundantly stored many times as part of different traces in the trace cache. A trace is typically limited by its size unless it is terminated by the occurrence of an indirect jump, procedure call or return. Traces are fetched from the trace cache with the help of the next trace predictor and assigned to the dispatch stage of a PE. Trace level sequencing does not always provide the required trace to a PE. Instruction level sequencing is required to construct non existent traces or to repair mispredicted traces. There is a trace buffer assigned for each PE, that is informed to construct a new trace on a trace cache miss. The next trace predictor performs control speculation by predicting the traces to be executed concurrently. The trace ID is a combination of the address of the first instruction in the trace and the history of the branches in the trace. Value prediction and memory dependence prediction are used to achieve better performance. Selective

reissuing is performed in case of a data misspeculation (misspeculation on a register value, memory address, memory value).

Thread spawning scheme

Correlated predictor:

This architecture uses a correlated predictor to predict the next trace to be executed [27]. Similar to correlated branch prediction [28], this prediction scheme has a pattern history table (PHT) indexed by global path history information. The global path history is a combination of the previous traces leading to the current trace, and is stored in a shift register. Each entry in the PHT consists of a next trace ID, an alternate trace ID and a saturating counter, based on whose value the target is either predicted or is replaced with a new next trace ID.

It is possible that different path histories could lead up to the same PHT entry. This aliasing could lead to loss of accuracy. In order to reduce the impact of cold-starts and aliasing, the correlated predictor is augmented with a second smaller predictor that uses only the previous trace ID as its index, and not the global path history. Each entry in the correlated predictor is tagged with the last trace to use the entry. If the tag matches then the correlated predictor is used, otherwise the smaller predictor is used. If the counter of the smaller predictor is saturated its prediction is automatically used, regardless of the tag.

Return history stack: This architecture uses another scheme to increase the accuracy of its predictor. If a call is seen in the trace, the history information is kept in a stack called the Return History Stack (RHS). If a return is seen in a different trace, this history is popped from the stack and a part of this history is used along with the conventional global path history. With the RHS, when a subroutine

returns, the path history now has information about program flow before the call. This enhances prediction accuracy because, program flow after a subroutine return correlates strongly with program flow before the call to the subroutine. Without the RHS, information from the subroutine execution dilutes the history, as the history is constantly updated during run time. This scheme is particularly useful if the subroutine is long and will force out any pre-call information from the history register.

3.2.4 Dynamic Multithreading

Execution model

Dynamic Multithreading (DMT) [8] is based on an SMT architecture model, with multiple thread processing units on the same processor core. Each thread unit has its own PC, rename table and LSQ. The threads share the register file, memory hierarchy, execution units and branch prediction tables. A single program is dynamically divided into multiple threads, at loop and procedure boundaries, and assigned to execute on separate thread units. The basis for this model is that program behavior after a call strongly correlates with program behavior before the call. The same argument applies to backward branches of loops. This behavior allows concurrent thread execution without many data dependences. Value prediction is used to relax the limitations imposed by inter thread data dependencies. This architecture also proposed a novel speculative state hierarchy, where speculative instructions and results are stored in a slow large buffer outside the pipeline. In case of a misspeculation, affected instructions are fetched from this buffer and reexecuted.

Figure 3.4: Thread spawning in DMT

Thread spawning scheme

There are many interesting heuristics followed by this architecture for thread ordering and thread selection.

Thread spawning: This was one of the first schemes to speculate threads dynamically without using any compiler support. It uses instructions following loop exits and procedure calls to spawn speculative threads. A thread spawns a new thread at procedure calls (point A) or backward branches (point B) as shown in Figure 3.4. Default start address of the new thread is the instruction that follows the call or the backward branch instruction. A history buffer is used to predict after loop thread addresses that are different from the default values. A spawned thread is allowed to spawn further threads. The same thread is allowed to spawn multiple speculative threads. This flexibility allows threads to be created out-of-order, which makes the DMT thread spawning scheme very different from other

Figure 3.5: Thread ordering in DMT

schemes. With threads being created out-of-order, there is a need for a thread ordering scheme.

Thread ordering: Thread ordering in DMT works as follows. From the perspective of a thread, the most recent thread it spawns will be the earliest one to retire. An ordered tree is maintained to keep track of the program order of threads. Threads spawned by the same thread are inserted into a tree in order. Figure 3.5 shows thread ordering for the threads spawned in Figure 3.4. At any point in time the thread order is determined by walking the tree in that order, for example from top to bottom, and right to left.

Thread allocation: The thread allocation policy is pre-emptive. When all thread contexts are being used, a new thread earlier in program order pre-empts the lowest thread in the order list. The lowest thread is then squashed, all its state is reset, and its context is assigned to the new thread.

Thread selection: A thread is selected based on three parameters - 1.) Thread retirement 2.) Thread overlap and 3.) Thread size.

The thread start address is used to index into an array of saturating counters. The thread is selected based on the count. The counter is updated when the selected thread is retired or squashed. The counter is reset for a thread that is too small or does not sufficiently overlap other threads. When a thread is not selected because of its counters state, there is no execution of this thread and consequently no feedback information about the prediction accuracy. Without feedback, the threads counter is stuck in a not-taken state. To avoid this problem, the counter is also updated by observing the spawn points at retirement in reverse order to the join points to estimate how the thread would execute if spawned.

3.2.5 Speculative Multithreaded Processors / Clustered Speculative Multithreaded Processors

Execution model

Speculative Multithreaded Processor (SM) [10] is decentralized architecture with each thread unit (TU) having a local register file, rename table, execution units and ROB. It tries to exploit parallelism by creating threads on loop iterations and executing multiple iterations of the same loop concurrently on separate thread

units. There is only one fetch unit, that fetches one instruction each cycle. The same instruction is assigned to each thread unit for execution. Only those loop iterations that follow the same control flow and execute identical set of instructions are executed concurrently. The TU's execute in a circular ring with the head being the non speculative TU. Only one TU executes non speculatively and commits to the architectural state. All subsequent TU's execute speculatively. Only the non speculative TU can spawn a new thread. The non speculative thread spawns multiple speculative threads based on two parameters - 1.) The number of thread units available. 2.) The number of loop iterations that can be executed concurrently.

When a non speculative thread finishes all its work, it converges with the speculative thread. It commits its state and makes the next thread in the ring the non speculative thread.

The architecture shows good performance for the fetch bandwidth it uses, in comparison to a superscalar with the same fetch bandwidth. If one thread follows a different control flow then that thread unit and all the subsequent tasks are squashed. There are not many loops that can be exploited with this algorithm. Hence an improvement over this architecture was proposed as Clustered Speculative Multithreading (CSMT) [11]. The significant modification in CSMT is that, it has the capability to execute loops that follows different control flows. In their experiments they observe that the number of different control flows in different iterations of the same loop is very low. The CSMT architecture still fetches one instruction each cycle similar to the SM architecture. The fetched instruction is forwarded to all TU's that follow this control flow. However, the main difference is that, other TU's requiring a different instruction use the fetch unit in a round

robin order.

The loops are stored as traces and the next trace is predicted with the same algorithm used by Trace processors, treating loop traces similar to traces in Trace processors. Another key performance factor is that a non speculative TU starts from the innermost iteration of a loop and spawns threads moving outwards in the nesting order. This means that the new threads are outer iterations of loops and consequently larger. This gives very good load balancing features, limiting the time the speculative thread stalls waiting for the non speculative thread to finish execution.

Thread spawning scheme based on loop iterations

Both SM and CSMT spawn threads on loop iterations. The task is to identify occurrence of loops during dynamic execution and mark them out so that control logic can spawn threads upon occurrence of the same loop. A loop trace ID consists of the start PC of the loop combined with the outcomes of branches enclosed within that loop body. Dynamic loop detection is based on identifying backward branches.

The boundaries of loops are identified as follows [29]. All the information related to the loops are stored in a current loop stack (CLS). The CLS maintains entries characterized by the beginning and terminating instructions of the loop. To be specific, each CLS entry contains two fields -

- 1.) Entry T corresponding to the start PC of the loop body. A backward branch from the end of the loop should take control flow to T. T is the starting instruction for all the speculative threads. If a backward branch takes program execution to T, then the algorithm searches if T exists in the table. If T exists, it

means another iteration of the loop is starting. A saturating counter is updated accordingly.

2.) Entry B corresponding to the boundary of the loop, a backward branch at this PC leads control flow to T. It may be possible that different PCs may branch back to T. B corresponds to the highest of those backward branches. The loop body can include nested loop iterations or entire function bodies. If the loops are nested the CLS functions like a normal stack, pushing in outer loops further into the stack. The top of the stack always has the innermost loop.

A loop execution table and a loop iteration table are maintained to aid in making more accurate predictions. Loop execution information is related to how many times a loop executes overall. Loop iteration information is related to each loop iteration, the data dependences and the predictability of data dependences and data values.

Generic thread spawning scheme

The thread spawning techniques discussed so far extract thread level parallelism (TLP) making use of constructs that are native to the application program like loop iterations, loop continuations and subroutine continuations. Gonzalez et al. [30] showed that by creating threads only on the basis of program constructs a thread spawning scheme exploits very little parallelism available in programs. They showed that any basic block can be a good point to fork tasks, not just loops and procedures. They used the CSMT execution model for their experiments.

This technique profiles sections of the code from the CFG and finds out potential points to fork tasks. Basic blocks in the CFG are treated as nodes and connected together based on the control flow. The edges between nodes are weighted

based on how many times control flows through those edges. Less frequently visited edges are trimmed and absorbed into the neighbouring nodes and corresponding changes are made to the control flow of that node.

Once the CFG is constructed, a table is created with nodes having spawning points as rows, and nodes having reconvergence points as columns. The reaching probability, which is the probability of program flow starting from the spawn point and merging at the reconvergence point is entered for each row and column. This reaching probability is the sum of frequencies from a spawn point until a reconvergence point. The only constraint is that a particular node can be a spawn or reconvergence point for only one thread. The same node can feature as the intermediate point in any number of threads. A pair of nodes are set as the spawn point and reconvergence point based on a set of rules. These rules are similar to that followed by DMT, which are 1.) Thread retirement 2.) Thread size 3.) Thread overlap due to data independence and value prediction.

Summary

Let us try to put together the details presented on the prior work. The key to any SpMT architecture is to start a new thread at a point which is sufficiently distant from the current execution point. This can guarantee sufficient parallelism between threads. It is also a good idea to start a thread at a point which is CI of the spawning point, to avoid local branch mispredictions from affecting other tasks. The mechanisms that can detect control independent points far away from the current instruction rely on program constructs [8, 10, 11], giving very little flexibility to spawn threads that are load balanced. There are other mechanisms that detect CI points at consistently similar distances from the targeted branch

[13, 27]. However, they tend to be of very short distances to be of any use in an SpMT architecture. We need a mechanism that can detect CI points at consistently similar distances from the spawning point, yet sufficiently distant from the spawning point to facilitate parallelism. This thesis is the first proposal of such a task predictor, which will be explained in detail in the next chapter.

Chapter 4

Task Predictor

This chapter begins with an overview of the task predictor. It is followed by the implementation details of the task predictor. This chapter ends with an overview of the architecture on which the task predictor has been tested.

4.1 Task Predictor Overview

4.1.1 Base Predictor

The predictor that we use in our model derives largely from the dynamic reconvergence predictor suggested by Collins et al. [13]. We will call it the base predictor. The important characteristics of this predictor are:

- This is a hardware scheme and does not rely on compiler support.
- It does not rely on program constructs like loop iterations and procedure calls to find out reconvergence points.

The concept is based on a profiling scheme where a summary of the expected control flow from the targeted branch to the reconvergence point is constructed by observing the committed instructions in the dynamic instruction stream. Potential reconvergence points for each category (discussed in section 2.1) are identified for each targeted branch. Since the algorithm is based on run time information, it is very accurate in predicting reconvergence points correctly. It maintains a back end structure updated at the commit stage, called the reconvergence prediction table

(RPT). Each entry in the table, called an RPT entry, contains the targeted PC along with its potential *reconvergence below max PC*, *reconvergence above max PC* and *rebound reconvergence PC* values. An RPT entry also stores some additional information so that control logic can decide on the best reconvergence point, if the targeted branch has multiple reconvergence points. Each branch keeps track of its call level and only branches belonging to the latest call level are trained for reconvergence (because of the reason mentioned in section 2.1).

There was however one aspect of this algorithm that needed to be modified so that it could be used in an SpMT architecture. The predictor attempts to identify reconvergence points as close to the targeted branch as possible. Their results show that the reconvergence point lies within 16 instructions of the targeted branch for 80% of the branches. [20] also showed that exposing branches to their precise CI points in programs mostly will lead to very small threads. Therefore the base predictor is suitable for an SMT like architecture where multiple threads compete for centralized resources. As explained in section 2.2, we would like the reconvergence point to be many instructions away from the targeted branch, so that there is enough parallelism between threads to be exploited on multiple cores. The base predictor is modified as explained below to make it capable of predicting such reconvergence points.

4.1.2 Enhancements on the Base Predictor

Predicting Large Tasks

In the modification done on the base predictor we try to exploit the transitive property of reconvergence points. Figure 4.1 shows part of a program flow. B1 is

Figure 4.1: Concept of linking reconvergence points

a branch and instruction R1 is its reconvergence PC. B2 is the first branch that follows R1, and R2 is B2's reconvergence PC. Extending the same relationship, B3 is the first branch that follows R2, and R3 is its reconvergence PC. It is clear that in addition to R1, even R2 and R3 are also control independent of B1. So R3 can be used as the reconvergence point for B1.

The logic that needs to be augmented to the basic predictor is as follows. After finding out the reconvergence point of a branch, the control logic needs to track the first branch that will follow. We will call this branch the *link PC* (LPC) of the targeted branch. Using LPC, it is possible to chain reconvergence points together to get a final reconvergence point that is many instructions away from the targeted branch, as compared to the reconvergence point predicted by the base predictor.

Updating RPT from Multiple Cores

The base predictor was proposed for a single core multithreaded processor, with one retirement stream. The control logic in the base predictor needs to be extended to allow the table to be updated from multiple cores. The additional rules that are introduced are as follows.

Each branch, in addition to keeping track of its call level, also keeps track of the core that is training it. Each branch is uniquely associated with one core, as long as the core is active. As long as a branch is associated with one core, no other core can train that branch. When a core is retired or squashed, it releases all the branches associated with it. After that, the first core to execute this orphaned branch will associate itself with the branch and continue training it further. Within the same core, call levels are tracked similar to the base predictor. If the same branch executes at different call levels in different cores, the earliest core to see the branch will take ownership of the branch and train it. All other cores will not attempt to train that branch, even if they are on different call levels.

4.2 Task Predictor Implementation

The task predictor block diagram is shown in Figure 4.2. The task predictor hardware consists of:

- A reconvergence prediction table (RPT) which stores branches along with their reconvergence PC's and other information that will aid in predicting tasks with good characteristics.
- An active reconvergence table (ART) that stores branches corresponding to their call levels in the form of a stack. Only branches at the top of ART will

Figure 4.2: Task predictor block diagram

be trained for reconvergence.

- A controller to observe the retirement stream and update the RPT.

The RPT is indexed by the address of the targeted branch whose reconvergence PC is to be predicted. The table contains information on the observed dynamic behavior of each branch. An example of the base predictor RPT is shown in Figure 4.3.

All the information corresponding to a branch is collectively called the RPT entry for that targeted branch. An RPT entry contains the following fields:

- *TPC*: The address of the targeted branch whose reconvergence PC is to be predicted.
- *BelowPotential (BP)*: The below potential reconvergence PC.

Tag	AP	AP	RP	RPC	A	cnt_r	cnt
2000	3100	0	2002	3100	0	248	368
2408	5398	898	3602	5398	1	16	32
3204	0	0	2802	0	1	0	0
0	0	0	0	0	0	0	0
5445	9900	0	7296	7296	0	36	498
7300	7302	1982	7302	1982	0	162	162
0	0	0	0	0	0	0	0
9932	9936	0	9936	0	1	0	0

Figure 4.3: Reconvergence prediction table for the base predictor

- *AbovePotential (AP)*: The above potential reconvergence PC.
- *ReboundPotential (RP)*: The rebound potential reconvergence PC.

The term 'potential' refers to the current best guess of a potential reconvergence PC, one for each category i.e. *rec below max*, *rec above max* or *rebound reconvergence* PCs.

- *active*: There is one active bit for each potential. When a targeted branch is executed, all the active bits are set. The active bit being set indicates that the targeted branch has been observed and the corresponding potential is ready to be trained in response to committing instructions. The active bit corresponding to each potential will be cleared if that potential is either matched or updated, and will remain cleared until the targeted branch is executed again.
- *RPC*: The reconvergence PC.

Each RPT entry also contains additional information that will aid in creating tasks with good characteristics.

- *RPC_count*: The number of instructions from TPC to RPC.
- *max_RPCcount*: The maximum number of instructions from TPC to RPC.

4.2.1 Training Algorithm

Each RPT entry is trained in multiple training phases. Until a branch instruction executes, its RPT entry stays inactive. When a branch executes its RPT entry is made active. A branch's RPT entry is updated only by the committing instructions in the same call level. As explained before it is important to track call levels because, it is possible that a branch can hit a return before it reaches its expected reconvergence PC, in which case it will reconverge at a totally different PC, and the algorithm needs to take this behavior into account. When a new call level is entered, all branches belonging to the earlier call level are temporarily inactivated. They are activated only upon the return instruction which will take execution back to that call level. It should also be made clear that the same committing instruction can train multiple branches in the same call level. Each RPT entry tracks its call level and trains itself accordingly. It needs to be ensured that only branches in the latest call level train themselves in response to the committing instructions. For this purpose, RPT entries are organized based on their call levels in a structure called the active reconvergence table (ART). The ART works exactly like a stack. When a call occurs, the RPT entries corresponding to the old call level are pushed one level down. RPT entries corresponding to the new call level will be on top of the stack. When this procedure returns, these RPT entries are popped out from the ART. Only RPT entries at the top of ART will be in training, since these correspond to the latest call level. All the branches seen until then in the latest call level will train simultaneously in response to committing instructions, if any of

their active bits are set. It is possible that the same branch exists in different call levels. In such a case, the same RPT entry will get updated in multiple call levels.

The reconvergence point for a branch is detected in two steps:

- 1.) Program behavior is observed to find an appropriate candidate for each category of potential reconvergence PC for every targeted branch.
- 2.) The best reconvergence PC is selected from these categories based on a set of rules.

Out of the four categories return reconvergence is determined by the top of the return address stack (RAS) when a return instruction is observed. For the rest of the categories, the rules to update the reconvergence potentials are as follows.

rec below max convergence:

1.) When a branch is executed for the first time, the *BelowPotential* is initialized to the sequentially next PC following the branch. 2.) On all future executions of the branch, the *BelowPotential* becomes active (its active bit is set), and may be updated in response to committed PCs. 3.) If the *BelowPotential* matches the committed PC, the active bit is cleared. 4.) Else if a PC below the *BelowPotential* is committed, *BelowPotential* is updated with the committed PC and the active bit is cleared.

One of the implications of the above training algorithm is that the *BelowPotential* can only be updated due to a taken forward branch. It converges quickly, because every execution of the branch results in either a correct prediction (when the reconvergence PC is executed), or an unexpected control flow which leads to a forward branch over this point, causing an update with a better prediction. Thus,

the total number of reconvergence mispredictions for a single branch is bounded by the actual control flow, typically a small number.

rec above max convergence:

1.) This is trained similar to *BelowPotential* except that it is updated only in case of a PC above the TPC and below the current *AbovePotential*. 2.) When a branch is first executed, *AbovePotential* is initialized to an invalid value. 3.) Active bit corresponding to *AbovePotential* is set when the TPC executes. *AbovePotential* is updated by the first instruction executed above TPC. 4.) If the *AbovePotential* matches the committed PC, the active bit is cleared. 5.) Else if a PC below *AbovePotential* is committed, *AbovePotential* is updated with the committed PC, and the active bit is cleared.

rebound reconvergence:

1.) Initially and each time the *BelowPotential* is updated, *ReboundPotential* is set to the sequentially next PC following TPC. 2.) After *BelowPotential* executes, program execution is observed for any PC that executes below TPC, below *ReboundPotential* but above *BelowPotential*. 3.) If such a PC commits, *ReboundPotential* is updated with the committed PC, and the active bit is cleared. 4.) If *ReboundPotential* matches the committed PC (before or after *BelowPotential*), the active bit is cleared.

In some cases, a branch may hit a return instruction before it reaches its expected reconvergence PC. In some cases, each branch can have multiple potential reconvergence PCs. For example, a branch can have both its below potential and

rebound potentials as valid values. We will need a set of rules to decide on the best reconvergence point. Hence, each RPT entry will also include two other fields to decide the best reconvergence PC.

- 1.) *HitReturn*: A *HitReturn* bit corresponding to each potential reconvergence PC is maintained. Initially this bit is cleared indicating that the reconvergence PC is reached before a return statement is observed. If a return instruction is seen, then *HitReturn* is set.
- 2.) *ReachedFirst*: The control logic may identify more than one valid reconvergence PC for a targeted branch. To decide which is the best RPC, a *ReachedFirst* bit is maintained for each potential RPC. Initially these bits are cleared indicating that the RPC which will be reached first is unknown. When we see the first potential RPC, its corresponding *ReachedFirst* bit is set to indicate that the particular RPC is reached prior to the other two categories. In this way, if one RPC is always reached first, ultimately its *ReachedFirst* bit will always be set.

In summary, the rules to predict the best RPC are as follows: 1.) If *HitReturn* is set, then the instruction to which the procedure return leads to (from top of RAS) is the RPC. 2.) If *ReachedFirst* is set, then that potential RPC is the RPC. 3.) If the above two conditions are not satisfied, *BelowPotential* is the best RPC.

4.2.2 Linking

The process of linking to form larger tasks is explained below. The RPT will need to be augmented with further information to build larger tasks. Each RPT entry is extended with another field that contains the LPC. This represents the first branch

encountered after the matched RPC or updated RPC is committed (as explained in section 4.1.2). The LPC will have an RPT entry of its own, where it will be the TPC, with its own RPC and LPC.

The table is also extended with fields containing corresponding information from TPC to LPC (similar to information from TPC to RPC).

- *count*: The number of instructions from TPC to LPC.
- *max_count*: The maximum number of instructions from TPC to LPC.

4.2.3 Updating RPT from Multiple Cores

To update the RPT from multiple cores, each RPT entry is augmented with the core identifier. During execution from TPC to RPC, it is possible that the call level can change. The spawned core needs to know the call level of the instructions that are assigned to it. In hardware, a call level predictor needs to be implemented. For our simulations we use an Oracle model (explained in section 5.3) to inform the new core its call level at the time of spawning.

The RPT now has targeted branches with their reconvergence points and links. The next step is to execute the tasks from our task predictor on an SpMT architecture. The next section gives an overview of the architecture on which we have tested our predictor.

4.3 DOE Architecture

Figure 4.4 shows the Disjoint Out-of-Order Execution (DOE) execution model. The DOE architecture consists of a number of cores connected together by a ring

Figure 4.4: DOE execution model

network. Task predictor logic monitors the execution in each core and assigns tasks to each core. Each core fetches and executes instructions from its own task until it reaches the end of its task. Tasks are spawned in program order making data communication simpler. The window of instructions in the dynamic execution range is bounded by the first fetched instruction in the oldest task and the last fetched instruction in the latest task. However, instructions within the execution range are fetched, executed, and retired out of order. The program order of tasks matches the executing cores physical order in the ring. A head pointer and a tail pointer rotate around the ring and are used to allocate cores to tasks and to commit tasks in program order.

Each core has unique characteristics that make it suitable for DOE execution

which are explained below.

Disjoint data threads execution:

Each core executes two disjoint data threads out-of-order. One data thread is formed of all instructions that are data dependent on a previous active task, and the other data thread is formed by all the instructions that are data independent of previous active tasks. The independent data thread is executed immediately when a task is dispatched, while the dependent data thread is buffered outside the execution pipeline and executes when the previous task completes and commits. By this time all previous mispredicted branches have been corrected and input data propagated from the previous task. The dependent data thread therefore does not block the execution of the independent thread. This achieves two goals: a) it supports control independent execution of tasks, and b) it provides tolerance to the delays encountered on input data produced by other cores and communicated through the ring.

Resource efficiency:

The non-blocking latency tolerant cores are built without large multi-ported cycle critical buffers. Data dependent threads free all core resources and wait in special SRAM memory outside the execution pipeline, where they wait until their inputs become available. Independent threads execute and pseudo-retire freeing core resources as well.

Checkpoint processing and recovery:

The core architecture utilizes checkpoints [16] for resource efficient recovery

and for instantaneous integration of results from multiple data threads. By using checkpoints for recovery, completed independent instructions can be pseudo-retired and removed from the pipeline to free resources. In case an exception or misprediction is detected in the dependent data thread, precise state can be restored to a checkpoint and execution restarts after flushing the pipeline and squashing the task. If the data dependent thread is completed without exceptions or branch mispredictions, the results from the dependent and independent threads are integrated by merging state from two different checkpoints.

This chapter explained the task predictor and the architecture on which it has been tested. Testing the predictor would involve spawning, execution, retirement and squashing of tasks. The next chapter will explain the simulation setup, experiments and results.

Chapter 5

Simulation Methodology and Results

This chapter starts with an overview of the tool used to simulate the task predictor and the DOE architecture. It is followed by details on the simulation methodology along with experiments and inferences. In section 2.2, we discussed about task size, control independence and data independence being the key factors influencing task selection. Handling task size and control independence requirements are explained in section 5.2. Handling the data independence requirement is explained in section 5.3.

5.1 Simulation Environment

We use an execution driven performance simulator derived from the PTLsim infrastructure for our experiments. PTLsim is a simulator that models pipeline execution delay and cache latency accurately. PTLsim simulates x86 code after converting complex instructions into RISC-like micro-ops (uops), a technique used in Intel and AMD processors. Modifications are made on the base PTLsim code to model a multicore architecture with each core modeled as an in-order single issue processor. All the cores share the L1 instruction cache and L1 data cache. The L1 instruction and data caches are multi-ported and can be accessed simultaneously by all cores. The cores also share the Miss Status Holding Registers (MSHR) [31]. L2 and L3 caches are unified instruction and data caches. Cache sizes and associativity are listed in Table 5.1. The pipeline and branch predictor configurations are also listed in Table 5.1. The simulator is also augmented with the task predictor

Base Core	8 stage, 1-wide, In-order
DOE Core	12 stage, 3-wide decode, 1-wide issue
L1 Data Cache	64 KB, 4-way, 2 cycles, 64-byte line
L1 Insn. Cache	64 KB, 4-way, 2 cycles, 64-byte line
L2 Cache	256 KB, 4-way, 12 cycles, 64-byte line
L3 Cache	2 MB, 8-way, 50 cycles, 64-byte line
L3 to Memory Latency	350 cycles
Branch Predictor	Combined bimodal and gshare 64K each bimodal, gshare 4K BTB

Table 5.1: Simulation setup

and the control logic to monitor the execution of tasks on the cores.

5.2 Task Spawning

This section explains our algorithm to spawn large and control independent tasks. Since we are in the initial stages of our design and we do not know what task size gives good performance, we pre-determine the size of the task we intend to spawn.

During the initial execution phase, there is only a single core running. A certain number of instructions are run to warmup the pipeline and prediction tables. After the warmup period, the executing core tries to find a point to spawn a task. When it encounters a branch, it probes the RPT to see if the branch has an RPT entry and a valid RPC. If it does, then the number of instructions from the TPC to RPC is noted. If the count satisfies the task size requirement, then the TPC

Tag	BP	AP	RP	RPC	A	cnt-r	mcnt-r	LPC	cnt	mcnt
2000	3100	0	2002	3100	0	248	368	3204	264	384
2408	5398	898	3602	5398	1	16	32	5445	63	79
3204	0	0	2802	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
5445	9900	0	7296	7296	0	36	496	7300	40	502
7300	7302	1982	7302	1982	0	162	162	2000	180	180
0	0	0	0	0	0	0	0	0	0	0
9932	9936	0	9936	0	1	0	0	0	0	0

Figure 5.1: Linking to create a task of size greater than 1000 instructions

is marked as the spawn point and RPC as the join point. If this TPC-RPC combination does not satisfy the task size requirement (which is typically the case), then the task size is increased by creating links (as explained in section 4.1.2). The process of linking RPT entries together continues until the task size is greater than the task size threshold. The RPC of the last RPT entry that is probed is considered the *join PC* (JPC). The branch instruction that was started with is considered the spawn PC or *fork PC* (FPC). (The terms *spawn* and *fork* can be used interchangeably.)

Figure 5.1 shows the linking process in an RPT, aiming for a predetermined task size greater than 1000 instructions. *cnt-r* refers to the current observed count from TPC to RPC, while *mcnt-r* refers to the maximum count from TPC to RPC

observed so far. Similarly *cnt* refers to the current observed count from TPC to LPC, while *mcnt* refers to the maximum count from TPC to LPC observed so far. Consider the branch at 2408 is encountered and is probed in the RPT to spawn a new task. Its RPC is only a few (32) instructions away from itself. But the LPC leads to a more distant reconvergence point ($79 + 498$ instructions). If branches with tags 7300 and 2000 are further linked, a large task size is formed ($79 + 502 + 180 + 368$), which is higher than the predetermined threshold of 1000 instructions. Hence the RPC (3100) of the latest probed TPC (2000) will be the join PC. It is interesting to notice from the above example that control flow can move in any direction, both above and below in the static code, in order to create tasks of larger sizes.

Now a new task can be spawned at this targeted branch on a new core, which will start executing instructions speculatively from join PC onwards, while the spawning task continues executing instructions from the branch onwards. The spawning task is called the parent and the spawned task is called the child. The child task can spawn further speculative tasks.

There are some points that need to be considered when RPT entries are linked to form larger tasks:

- 1.) If a TPC links to itself (i.e. for an RPT entry, LPC is the same as TPC), then the task predictor will loop in the same RPT entry until the task size is greater than the threshold. The task predictor is incorrectly led to believe that a large sized task can be created by following this RPT entry. In this case, when program execution is observed from FPC to JPC, it is seen that the parent converges very

Task Size	bzip2 (%)	equake (%)	mcf (%)	parser (%)	twolf (%)
0 – 500	9.52	0.02	0.02	1.84	58.85
500 – 1000	14.7	3.08	0.31	0.08	26.93
1000 – 1500	9.99	37.01	24.1	0.08	9.43
1500 – 2000	7.33	28.64	49.33	0	2.61
2000 – 2500	10.5	25.44	24.99	9.29	1.93
2500 – 3000	6.72	3.65	1	72.78	0.21
>3000	41.2	2.14	0.22	15.75	0.02

Table 5.2: Distribution of task sizes. Task size is in number of instructions.

soon. The algorithm needs to take care of this condition. So as a rule, control logic ensures that a TPC does not link to itself. If the first branch encountered after the RPC is the TPC itself, then the control logic treats this as a regular instruction (not a branch) and tries to look for the next branch to assign as the LPC.

2.) Another situation where a link may break is when the LPC does not have an RPT entry. This may happen when the RPT entry associated with the LPC gets replaced in the table due to shortage of memory space. In this case, the TPC-LPC combination is left hanging. So as a rule, task control logic needs to ensure that an LPC has an RPT entry before using the LPC to create larger tasks. If the LPC does not have an RPT entry, control logic terminates its attempt to fork at that FPC and looks for a different point to fork. This still means that the same RPT entry can be linked many times over in an attempt to form a large task, provided the link comes from a different RPT entry. For example, if A, B and C are RPT entries, $A \rightarrow A$ is not allowed, but $A \rightarrow C \rightarrow B \rightarrow A \rightarrow C \rightarrow B$ is allowed.

Table 5.2 shows the task size distribution produced by our task predictor for five spec2000 integer applications. The rows in the table show the percentage of

spawned tasks within a specific size range for each benchmark. We set the task size target in our predictor to a minimum of 1000 instructions. Tasks predicted to be less than 1000 instructions are not spawned by the task dispatcher. Even though our predictor is capable of highly accurate control independence prediction, the task size prediction is quite inaccurate and widely varies for all benchmarks. The variation in the task size has a big impact on performance due to load imbalance, even under perfect parallelism conditions, as we show in section 5.6.1. Notice that even though the predictor is set to dispatch tasks with a minimum size of 1000 instructions, many spawned tasks are actually smaller due to task size mispredictions.

At this point, we have tasks of large size, distant and control independent from prior tasks. Task execution will be considered in the next section.

5.3 Task Execution

This section explains the algorithm to address the data independence requirement of tasks. In order to isolate the effects of data dependencies on the performance of our model, we augment our predictor with an *oracle*. This oracle acts as a perfect value predictor for the child task and ensures that the child task can proceed in parallel with the parent task without stalling due to inter task data dependencies.

The oracle serves three purposes:

- It mainly acts as a perfect value predictor giving the child core the exact register and memory state at the start of its execution.

- It isolates the impact of CI predictor accuracy on processor performance by comparing the difference between the perfect CI predictor and a conventional CI predictor.
- It also informs a speculative core of its call level at the time of spawning (as explained in section 4.2.3).

The oracle takes in the register and memory state at the time of forking and proceeds with program execution until the reconvergence point is reached. There are two possibilities here.

1.) *Prediction is correct:* The prediction is correct and the reconvergence point is hit. In this case, the oracle will have the correct register and memory state until the reconvergence point. The oracle communicates this information to the spawned core. Hence a new core can start executing in parallel with its parent, at the control independent point with the exact register and memory state without any inter task data dependencies. The speculative core uses local buffers to save its state.

2.) *Prediction is incorrect:* In the second case, when the control independence predictor misspeculates, the oracle is designed to work in two modes - *conventional reconvergence prediction mode* and *perfect reconvergence prediction mode*.

In the conventional prediction mode, the oracle does not convey the misprediction to the control logic and allows the child task to proceed and fork further tasks. At a later stage, when it is found that the prediction was wrong, all the work done by the child task and its subsequent tasks are squashed.

Benchmark	Accuracy (%)	Performance Drop (%)
bzip2	98.32	0
equake	100	0
mcf	97.24	7
twolf	96.1	25.8

Table 5.3: Impact of predictor accuracy on performance

In the perfect prediction mode, the oracle immediately informs the control logic that this prediction will not converge. Following this, the predictor will look for a different point to fork.

This approach gives us a convenient method to isolate the effects of predictor accuracy on performance as shown in Table 5.3. It is interesting to notice that in twolf, a 3.9% drop in predictor accuracy results in a 25.8% drop in performance. This is because of throwing away all the incorrect work done by the subsequent non speculative cores following a misprediction.

5.4 Task Retirement

After a parent spawns a child task, the parent tracks program execution in its own core to see if it converges with the child task. There are multiple ways in which the parent can decide that it has converged with the child task. Table 5.4 compares the accuracy (in percentage) and correlation between expected and observed task sizes using three different methods to track convergence.

1.) *No path info* - In the simplest method, the parent can decide that it has converged the very first time it encounters the JPC. In many cases, the same RPT entry is linked many times over while forming a large task. Consequently, the

Benchmark	No path info			Unique PC's in path			Exact path info		
	Acc	Exp	Obs	Acc	Exp	Obs	Acc	Exp	Obs
bzip2	94.7	1321.6	764	96	1197.3	721.7	89.3	1669.5	1588.3
equake	99.7	1040.4	459	84.9	1008	790	64.3	1004.3	1370.2
mcf	100	1106	2269.5	100	1070	2280	93.2	1078.7	2916.7
parser	100	1137.1	312.5	97.9	1040.3	561.4	27.8	1084.1	1936.6
twolf	99.6	1005.8	115.6	99	1006.5	323.6	94	1206.7	1801.2

Table 5.4: Tracking convergence using three different methods.

Key : Acc - Predictor accuracy (%), Exp - Expected task size (number of instructions, Obs - Observed task size (number of instructions).

JPC will be encountered many times before the task finally converges. So it is a naive approach to decide on reconvergence based on the first occurrence of JPC. Though this method predicts accurately, the main disadvantage of this approach is the disparity between the intended and observed task sizes. This disparity is obvious because, when FPC and JPC are predicted, in most cases the linking logic will iterate over the same JPC many times before convergence. But during actual execution if it is decided that a task converges on the first occurrence of JPC, it will miss out on the iterations which were considered to increase the task size when the task was predicted.

2.) *Exact path info* - Instead, it is a good idea to trace the path from FPC to JPC, noting the RPC's and LPC's that feature along the path, at the time of prediction. This method gives very good correlation between intended and observed task sizes, but predictor accuracy is very low.

3.) *Unique PC's in path* - It is sufficient to note only the unique PC's from FPC to JPC. However, the order in which they occur needs to be captured. This

order of occurrence is the key to the algorithm. This method gives good correlation between intended and observed task sizes, along with good predictor accuracy.

Since the *Unique PC's in path* method gives a good balance between accuracy and correlation between expected and observed task sizes, it has been used for all further experiments. During actual execution, if the RPC's and LPC's are observed in the same order, all the way until the JPC is hit, control logic decides that the task has converged.

In conventional branch prediction, a mispredicted branch can be detected at the commit stage. This is because, the condition over which the control logic had speculated, will be resolved by the time the branch reaches this stage. Since the branch instruction is always guaranteed to reach the commit stage, detecting branch misprediction is very deterministic and straight forward. But this is not the case with reconvergence prediction. The RPT does not store the exact number of instructions, but only the maximum and average number of instructions from the FPC to JPC. The exact number of instructions after which the control logic will find out a reconvergence misprediction is not known. So control logic establishes a threshold, within which it is expected to see the JPC. If this does not happen, it decides that the control flow has gone in the wrong direction. This threshold is decided as a tradeoff between prediction accuracy and misprediction penalty. A larger threshold would mean a longer wait time and more accurate predictions, but it would also mean more wasteful work in case of a misprediction. Typically this threshold is set at twice the task size. Table 5.5 shows the impact of wait time (as a multiple of expected task size) on predictor accuracy (in percentage)

Bmark / Wait time		1	1.5	2	2.5
bzip2	Acc	98.32	98.32	98.32	98.32
	Perf	1.01	1.01	1.01	1.01
equake	Acc	41.4	93.22	100	100
	Perf	1.16	1.98	2.71	2.71
mcf	Acc	47	88	97.24	97.24
	Perf	1.58	2.82	3.36	3.30
twolf	Acc	85.1	92.2	96.1	96.7
	Perf	1.78	1.86	2.04	2.02

Table 5.5: Impact of wait time over predictor accuracy and performance.
Key: Acc - Predictor accuracy (%), Perf - DOE performance (speedup of 4-core DOE over single core)

and performance (speedup of 4-core DOE over single core - see section 5.6.1). It can be noted that, as the wait time increases the predictor accuracy increases, but beyond a certain point, performance begins to decrease. This is because of more instructions being thrown away on a misprediction.

When a nonspeculative core converges it passes over the nonspeculative state to the next core and goes idle. It is now available and awaits its turn in the ring to execute the next task. The next core now becomes the head. When the parent task converges with its child task, it means that the speculation from the task predictor was correct. This also means that the work that has been done so far by the speculative core is correct. The speculative core is now made the non speculative core. The local state of the speculative core is committed to the architectural state. If the speculative core finishes before its parent (the parent could be either speculative or nonspeculative), it will have to stall.

5.5 Task Squashing

If the task predictor mispredicts, the child task needs to be squashed. There are two options to deal with the subsequent tasks that follow the child. In a simpler implementation they can be readily squashed which makes them immediately available to spawn further tasks. In a more complex implementation, the subsequent tasks can wait until the corrected task tries to fork again. If the corrected task spawns at the same point and the same mask is sent from the task predictor, there is no need to squash any of the subsequent tasks. The task predictor can be biased such that under this condition, it will attempt to use the same branch that it had used for spawning on the previous occasion.

Only the first implementation has been tried out in our experiments. A task misprediction can occur only in the conventional reconvergence prediction mode, the results of which are presented in section 5.3.

5.6 DOE Experiments

This section explains the experiments specific to the DOE execution model explained in section 4.3.

5.6.1 Ideal Performance

Figure 5.2 shows 4-core DOE speedup over a single-issue in-order core. The benchmarks mcf, equake and twolf show good performance of up to 3.5 for mcf. The performance however falls short of the peak performance of 4 which is what can be theoretically achieved with 4 cores. This is under the assumption of perfect parallelism between tasks (equivalent to perfect task input value prediction). The

Figure 5.2: Speedup of 4-core DOE

other two benchmarks, parser and bzip2, show small or no speedup. The key insight from this experiment is that load imbalance due to size variation between tasks can significantly limit SpMT performance, even with perfect parallelism and with no inter-task data communication delays. To explain the performance variation among the five benchmarks in this experiment, we show in Figure 5.3 the average core activity for 4-core DOE configuration with perfect parallelism. The graph shows the percentage of time a core is running, stalling, or idle with no task assigned to it. The performance problem of parser and bzip2 is a result of very high idle time. We often see long stretches of execution with the control logic unable to spawn a new task due to the predictor failing to predict a task with the targeted task size of 1000 that was set in the predictor.

Figure 5.3: Average core activity, 4-core DOE

5.6.2 Scalability

Since our ultimate goal is to achieve performance comparable to out-of-order superscalar processors by using multiple simple in-order cores, we measure potential DOE performance as the number of cores increases. Figure 5.4 and Figure 5.5 show that performance scales reasonably well with the number of cores. Since these experiments are with perfect parallelism, we can only conclude that on well performing benchmarks there seem to be no serious scalability issues up to 8 cores due to task size variability.

5.6.3 Performance for Various Levels of Parallelism

So far, our experiments conducted with perfect parallelism indicate that there is a good potential for performance with DOE, even on programs with very complex control flow that lead to wide variations in task sizes. The question now

Figure 5.4: DOE mcf speedup vs. number of cores

Figure 5.5: DOE twolf speedup vs. number of cores

Figure 5.6: 4-core DOE twolf performance for various levels of parallelism

becomes what is the potential performance achievable with DOE at various levels of parallelism.

We show in Figures 5.6, 5.7 and 5.8 DOE speedup of twolf under various levels of parallelism and for 4, 6 and 8 core configurations. In these figures, pn represents a simulation with n% parallelism between tasks. For example, p100 represents perfect parallelism or perfect inter-task value prediction, therefore no dependent execution is required, and p80 represents a situation where 20% of the task execution time is spent on dependent execution. We also show in the graphs for comparison the performance of 4 and 8-wide out-of-order superscalar execution. In Figures 5.9, 5.10 and 5.11 we show similar results for mcf.

The key insights from these graphs are the following:

Figure 5.7: 6-core DOE twolf performance for various levels of parallelism

Figure 5.8: 8-core DOE twolf performance for various levels of parallelism

Figure 5.9: 4-core DOE mcf performance for various levels of parallelism

Figure 5.10: 6-core DOE mcf performance for various levels of parallelism

Figure 5.11: 8-core DOE mcf performance for various levels of parallelism

- DOE latency tolerance helps maintain a good level of performance for moderate levels of dependent execution. For example, 30% of dependent execution in twolf on 4-core DOE configuration results in a reduction in speedup of 20% from the ideal case of perfect parallelism.
- Beyond a certain threshold that varies between benchmarks and configurations, performance suffers a significant hit. This threshold is 35% dependent execution or 65% parallelism (p65) on 4-core DOE configuration. The performance threshold moves closer to p100 as the number of cores increase.
- DOE performs close or better than 4-wide out-of-order execution on 6-core DOE if dependent execution is kept at 15% or better.

Chapter 6

Conclusions

6.1 Comparison with Prior Work

This section makes a qualitative comparison between the thread spawning schemes discussed in section 3.2 and our scheme.

The Multiscalar and Superthreading architectures follow a compiler based thread spawning scheme, requiring recompiling of existing binaries. In the compiler based scheme, memory dependences between instructions are not readily apparent and in order to disambiguate them, the instructions need to be decoded. Hence the compiler cannot do this disambiguation at the time of creating tasks. In the Multiscalar approach, even though it is difficult to disambiguate memory dependencies, loads are still speculated, to make sure the execution model does not lose out on performance. Memory dependences are tracked by means of an ARB, which keeps an account of all the loads and stores in all the tasks and their order. The ARB is a very complex structure to implement and moreover, in the event of a misspeculation a huge penalty is paid in the form of squashing the violating task and all subsequent tasks. This is a big limitation on the speculation throughput of Multiscalar architecture.

To overcome this limitation, Superthreaded architecture takes a step backward choosing not to speculate, but instead enforce data dependences. Superthreaded architecture does not achieve considerable performance from its thread spawning

scheme because, 1.) It only spawn threads along loop iterations, which may be very restricted. 2.) It stalls on data dependences in an attempt to avoid hardware complexity by not performing data speculation.

The task predictor proposed in this thesis is dynamic and will not need any change in existing and legacy binaries. Since our prediction is based on run time information, it is more accurate. Moreover, we do not need to speculate on memory dependences that are difficult to disambiguate, at the same time not losing out on performance like the Superrthreaded architecture, because of the latency tolerant feature of the DOE execution model. DOE does not buffer speculative state like Multiscalar architecture. Instead it use checkpoints ensuring efficient resource utilization.

The Trace Processor gives very good load balancing features because of the consistent trace sizes. This architecture is organized on the basis of a trace cache. Storing traces in a trace cache necessitates redundant storage of dynamic code, reducing resource utilization. Since traces are of a small size, it is very difficult to exploit parallelism within such short traces as they will invariably be data dependent. Small traces also have the problem of the task start overhead being a significant part of the total trace execution time. Traces are also of a fixed size, giving less flexibility.

Unlike Trace processors, our predictor works well with regular cache structures without requiring redundant storage of code. Our predictor also creates threads of flexible and large sizes.

Even though the DMT thread spawning scheme was proposed for an SMT architecture, it can be used for a distributed architecture as well. But since the thread spawning scheme relies on program constructs like boundaries of loops and functions, load balancing can be a significant problem because of the inconsistent thread sizes. DMT follows a complex thread spawning scheme, thereby necessitating a thread ordering scheme that is not simple.

Our predictor does not rely on program constructs, thereby providing the flexibility to create well balanced tasks. The program order of tasks matches the executing cores physical order in the ring. Hence DOE follows a very simple thread ordering mechanism,

SM and CSMT processors spawn threads on loop iterations. This gives good performance on numeric code. However it has been observed that sequential applications have irregular patterns and are not loop intensive. In addition to this, they also lose out on performance due to load imbalance. The same architecture follows an alternate approach, by storing the frequency of execution of each basic block in relation to other basic blocks. This would need very large tables, resulting in a costly implementation.

Our predictor explicitly identifies reconvergence points and guarantees control independence between tasks. It achieves this with a very hardware efficient technique.

In comparison to all the prior architectures, DOE gives better performance in case of inter task data dependencies because of its latency tolerant architecture.

6.2 Conclusions

The objective of this thesis was to create tasks for an SpMT architecture. This is a very critical activity since it is the starting point and the bottleneck that will decide the performance of any SpMT architecture. We started with the known problems of task creation for an SpMT architecture being, task start / commit overheads, load imbalance, inter task control misspeculations and inter task data misspeculations. We tried fixing each problem, knowing very well that each problem could present contradicting requirements. The initial goal was to predict control independent threads, without focussing on predicting large sized threads. With this approach we were able to predict threads that were consistently of similar sizes. Then we tried to look into the task size requirement. We used the transitive property of control independent points to them together to get larger tasks. Since the tasks were still of consistent sizes they gave good load balancing features. The inter data dependence requirement was handled with an oracle machine or a perfect value predictor. There are some encouraging signs because some of the experiments that we conducted have shown that most of the data dependences (80%) often tend to repeat over a period of time and hence they are value predictable. Including data dependencies into predicting tasks is left as future work.

We tried testing our predictor on a novel SpMT implementation, the DOE architecture, with in-order cores having latency tolerance and check point processing features. Another reason not to prioritize data dependences in our task prediction scheme is the latency tolerance feature of our DOE architecture. Our results

have shown that even with 20% inter task data dependences a 4 core DOE will clearly outperform an 8 wide superscalar. This is considering the fact that an 8 core DOE will occupy the same area as a 4 wide superscalar. There is tremendous potential for this architecture. There is still a long way to go in our research. The ultimate goal is to provide very good performance on sequential applications with latency tolerance, using an architecture that is power efficient and scalable. We will continue with our research to provide more answers.

6.3 Future Work

Our task predictor is still in its initial stage. This section details the potential for future work on our task predictor.

Improve the Task Predictor Accuracy:

The task misprediction penalty is high in an SpMT architecture, because all the instructions executed speculatively by multiple cores will need to be thrown away. The difference in the performance between the perfect task predictor and the conventional task predictor highlights the importance of the task predictor accuracy. It is possible to use more run time information to avoid forking on tasks that tend to mispredict. Saturating counters could be used for this purpose.

Reduce the Discrepancy between Predicted and Observed Task Sizes:

We observe from the results in Table 5.2, that even though we use the path information to decide on task convergence, there is still a considerable difference between predicted and observed task sizes. This discrepancy is critical because it disturbs the load balance and accounts for a lot of performance loss in the form

of stalled cycles. The RPT entry can be augmented with an *execution_count* field, which is a count of the number of times its TPC executes from the spawn point to the join point. This information is updated from the retirement stream.

Set the Task Size Threshold Dynamically:

In Figure 5.3 we see that in some benchmarks there are a lot of idle cycles because the predictor does not find a task that meets the task size requirement. This is because the task size is fixed statically. The task size threshold must be made dynamically adjustable based on run time performance.

Include Data Dependencies into Task Selection Criteria:

The tasks are currently being predicted without considering data dependencies between tasks. This is because of the oracle execution model. The data dependencies need to be predicted based on profiling information. Each RPT entry is augmented with two fields that hold information on register dependencies:

- *Set of influenced registers (SIR):* The set of registers that are expected to be written into from the spawn point to the join point.
- *Set of live-in registers (SLR):* The set of registers from the spawning task which will be used as inputs by the spawned task during its execution.

A memory dependence predictor is used to perform the same operations for memory dependencies.

This information can be used by the predictor to spawn tasks that have a high level of data independence.

Call level Predictor:

To update the RPT from multiple cores, the spawned core needs to know the call level of the instructions that are assigned to it. For our experiments we use an oracle model to communicate the new core its call level. A call level predictor needs to be implemented.

Optimization to Reduce Lookups on the Predictor Table:

The linking process needs to merge the RPT entries to avoid repeated lookups of the RPT. Each lookup of the RPT takes a lot of time in hardware.

Hardware Implementation:

Even though our performance simulator models the pipeline and cache hierarchy accurately, there is the need for a hardware model to study the cost of the implementation and power consumption.

References

- [1] J. E. Thornton, "Parallel operation in the control data 6600," *Proceedings of AFIPS Fall Joint Computers Conference, Vol. 26*, pp. 33–40, 1964.
- [2] R. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, pp. 25–33, 1967.
- [3] G. S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Transactions on Computers, Vol. 39*, pp. 349–359, 1990.
- [4] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," *In Proceedings of the 24th International Symposium on Computer Architecture*, pp. 206–218, 1997.
- [5] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *The 22nd International Symposium on Computer Architecture*, 1995.
- [6] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [7] G. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar processors," *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

- [8] H. Akkary and M. Driscoll, "A dynamic multithreading processor," *MICRO-31*, November 1998.
- [9] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors," *In Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997.
- [10] P. Marcuello, A. Gonzlez, and J. Tubella, "Speculative multithreaded processors," *In Proceedings of the 12th International Conference on Supercomputing*, pp. 77–84, 1998.
- [11] P. Marcuello and A. Gonzlez, "Clustered speculative multithreaded processors," *In Proceedings of the 13th International Conference on Supercomputing*, pp. 365–372, 1999.
- [12] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.
- [13] 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*, December 2004.
- [14] H. Akkary, K. Jothi, R. Retnamma, S. Nekkcalapu, D. Hall, and S. Shahidzadeh, "On the potential of latency tolerant execution in speculative multithreading," *IFMT-08*, 2008.
- [15] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," *In Proceedings of the 11th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2004.

- [16] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," *MICRO-36*, December 2003.
- [17] T. Vijaykumar, "Compiling for the multiscalar architecture," *Ph. D. Thesis, University of Wisconsin-Madison*, 1998.
- [18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," *In Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 25–35, 1989.
- [19] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of control independence in superscalar processors," *HPCA-5*, 1999.
- [20] E. Rotenberg and J. Smith, "Control independence in trace processors," *MICRO-32*, 1999.
- [21] Y. Chou, J. Fung, and J. P. Shen, "Reducing branch misprediction penalties via dynamic control independence detection," *ICS*, 1999.
- [22] C.-Y. Cher and T. N. Vijaykumar, "Skipper: A microarchitecture for exploiting control-flow independence," *MICRO-34*, 2001.
- [23] A. Gandhi, H. Akkary, and S. T. Srinivasan, "Reducing branch misprediction penalty via selective branch recovery," *HPCA-10*, 2004.
- [24] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. Akkary, "Transparent control independence," *ISCA-34*, 2006.

- [25] J. Tsai and P.-C. Yew, "The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation," *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 35–46, 1996.
- [26] E. Rotenberg, S. Bennett, and J. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," *In Proceedings of 29th International Symposium on Microarchitecture*, 1996.
- [27] Q. Jacobson, E. Rotenberg, and J. Smith, "Path-based next trace prediction," *In Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 14–23, 1997.
- [28] T.-Y. Yeh and Y. Patt, "Two-level adaptive branch prediction," *In Proceedings of 24th International Symposium on Microarchitecture*, pp. 51–61, 1991.
- [29] J. Tubella and A. Gonzalez, "Control speculation in multithreaded processors through dynamic loop detection," *HPCA-4*, pp. 14–23, 1998.
- [30] P. Marcuello and A. Gonzalez, "Thread-spawning schemes for speculative multithreading," *In Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, February 2002.
- [31] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," *The 8th Annual International Symposium on Computer Architecture*, pp. 81–87, 1981.
- [32] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, pp. 1609–1624, 1995.