Contemporary Engineering Sciences, Vol. 2, 2009, no. 11, 517 - 534

Dynamic Control Independence Predictor for Speculative Multithreading Processors

Walid J. Ghandour

Department of Electrical and Computer Engineering
American University of Beirut
Beirut, Lebanon
wjg00@aub.edu.lb

Abstract

This paper presents two algorithms. The first algorithm predicts future control independent points in the dynamic instruction stream. A control independent point is an instruction at which the program execution paths will converge regardless of the outcome or target of the current branch. We have developed this algorithm to use for dispatching tasks in a speculative multithreading architecture we are developing. Our algorithm significantly improves prediction accuracy over current predictor from which it is inspired. Moreover, while the best current algorithm predict the earliest dynamic instruction reachable before the next statically identified control independent point, our algorithm predicts true control independence. The second algorithm is used to construct large tasks suitable for our speculative multithreading architecture.

Keywords: Computer architecture, Control independence, Single chip multiprocessors, Speculative multithreading

1 Introduction

Control flow instructions such as conditional branches, indirect jumps, calls and returns, introduce control and data dependences that are difficult to deal with statically. High performance superscalar processors dynamically predict branches and

execute instructions along the predicted control flow path. Due to long pipelines, data cache misses and out-of-order execution, a processor often fetches and executes many instructions before it detects a mispredicted branch. In superscalar processors, correcting a misprediction involves squashing all instructions from the mispredicted path and restarting instruction fetch and execution from the correct path.

For each control flow instruction, regardless of the path taken, execution will eventually converge to a control independent block of instructions [10]. It is common for control independent instructions to be squashed as part of the mispredicted path, and then fetched and executed again along the correct path. The first instruction in a control independent block is called a reconvergence point, since the control flow eventually reconverges at this point, independent of the execution of the current control flow instruction.

Speculative Multithreading (SpMT) architectures [1][5][8][9][12][14][15] have many similarities but some key differences in the way they deal with branches as compared to conventional superscalar processors [11]. SpMT processors use control flow prediction to split a single sequential program stream into multiple threads, also called tasks in SpMT literature, that execute on multiple communicating scalar or superscalar cores, and in some proposals, on multiple communicating threads within a simultaneous multithreading superscalar processor. Although these tasks execute concurrently, they are explicitly ordered according to their position within the sequential program to which they belong. Data flows within each task as well as between tasks according to the sequential program order.

Each SpMT task handles branch mispredictions, as in superscalar processors, by squashing the mispredicted instructions and restarting the execution after correcting the mispredicted instruction address in the program counter (PC) register. However, since tasks are distributed across multiple logical or physical cores, SpMT provides an opportunity for exploiting instruction level parallelism (ILP) that is not possible within a conventional superscalar processor. If the control flow predictor in a SpMT processor predicts tasks correctly at future control independent points, then mispredicted branches can be handled completely locally within a task. Branch misprediction recovery does not require any corrective action in subsequent program tasks, since subsequent tasks are control independent and will be reached regardless of the previous control flow path. This is assuming that data is communicated from one task to another only after all previous mispredicted branches have been corrected.

We are currently working on a new implementation of speculative multithreading. The goals of our SpMT architecture are to use large tasks to reduce task dispatch and data communications overhead relative to the total execution, and to run these tasks on low complexity in-order cores. By using simple in-order cores to improve performance of sequential programs, our architecture avoids large, power hungry buffers and complex out-of-order execution hardware, such as multi-ported register files, register renaming, reservation stations and reorder buffers, all required in

modern superscalar architectures. Achieving our goals requires the ability to predict and dispatch large control independent tasks very accurately, in order to avoid squashing a task when branches that belong to previous tasks are mispredicted. The work we report in this paper has resulted from this need for highly accurate control independence prediction.

1.1 Paper Contributions

This paper makes the following contributions to dynamic control independence prediction:

1. It presents a qualitative analysis of limitations of the most general current dynamic control independence predictor [3] and provides examples of program control flow structures that reduce prediction accuracy.
2. It proposes a modified algorithm for control independence prediction. The new algorithm significantly improves the accuracy of the predictor.
3. The new prediction algorithm, after sufficient training period, eliminates false positives[1]. Our simulation model shows that execution always reaches a predicted control independent point.
4. It proposes an algorithm for constructing large tasks. This gives our SpMT architecture the advantage of small data communication overhead relative to task execution time.

The rest of this paper is organized as follows. Section 2 reviews related work and gives an overview of existing control independence prediction methods. Section 3 presents a qualitative analysis of the limitations of the best currently known prediction method using examples of common control flow structures for which the prediction fails. Section 4 describes our control independence predictor algorithm. Section 5 introduces the new algorithm that we use to construct large tasks. Section 6 presents our simulation methodology and results. We conclude in section 7.

2 Related Work and Current Control Independence Prediction Overview

Control independence [10] is a property that results from high level language constructs such as if-then-else, loops, and procedures. For example, the instructions following if-then-else statement, procedure return, or loop exit are control independent relative to the dynamic instructions and branches inside the if-then-else block, procedure or loop. For many compiler optimizations, a compiler identifies control independent points using the concept of a branch immediate post dominator

[1] False positives are predictions that execution control flow does not reach.

[4], which is an instruction following the branch that is reached on every path between the branch and the control flow graph exit.

Identifying control independence at compile time however has some limitations. Control independent points identified statically accounts for rarely or never executed paths, since a compiler lacks dynamic information. Second, it is often difficult or impossible to communicate control independence information to the hardware since this involves changes to the target Instruction Set Architecture (ISA).

A variety of hardware methods of various levels of complexity have been used by researchers for identifying control independence at run time. Dynamic Multithreading (DMT) [1] exploits well behaved loop and procedure structures and predicts that the static instructions following a loop backward branch or a procedure call to be reconvergence points in the control flow. DMT forks speculative threads at these points. The DMT prediction method is simple and almost always correct. However, the method identifies only a subset of control independence points, and for many applications, provides little or no freedom in choosing where the speculative threads are forked, resulting in serious load balance issues.

Other simple, heuristic-based methods for identifying and exploiting control independence include Skipper [2] and Selective Branch Recovery [6]. Both focus on simple if-then and if-then-else structures, but like DMT's loop and procedure heuristics, fail to identify many control independence structures.

The method of Control Quasi-Independent Points (CQIP) is a more aggressive approach for exploiting dynamic control independence [7]. Control Quasi Independent points are defined to be future instructions that have 95% or more probability of being on a future control path. This approach leads to higher misprediction rates than the simpler methods in DMT, Skipper and Selective Branch Recovery, and is less suitable for SpMT architectures that have high performance penalty on task mispredictions.

The work reported in [3] proposes a general technique that does not depend on the compiler or on heuristics to predict various forms of control independence. Unlike heuristics methods, [3] shows that the technique is "robust in the face of aggressive compiler optimizations", and reports prediction accuracy higher than 99% for SpecInt benchmarks [13] with different compilers and instruction set architectures. We will refer in the rest of the paper to the control independence predictor in [3] as CTW predictor, after the authors' initials.

The CTW control independence predictor identifies three major categories of control flow reconvergence: Reconverge Below Maximum, Reconverge Above Maximum, and Rebound Reconvergence. Fig. 1 shows control flow graph examples for these three major reconvergence categories. The definition of these three forms of convergence is the following:

1. Reconverge Below Maximum (RBM): this category refers to a branch for which the control independent reconvergence point in the program is below[2] the branch. Moreover, no instruction below the reconvergence point can appear between the execution of the branch and the execution of the reconvergence point. The point marked RP in the control flow graph shown in Fig. 1A is an example of Reconverge Below Maximum.
2. Reconverge Above Maximum (RAM): this category refers to a branch for which the control independent reconvergence point is above[3] the branch. Moreover, no instruction below the reconvergence point but above the branch can appear between the execution of the branch and the execution of the reconvergence point. The point marked RP in the control flow graph shown in Fig. 1B is an example of Reconverge Above Maximum.
3. Rebound Reconvergence (RR): this category refers to a branch for which the control independence point in the program is below the branch. Moreover, for at least one control flow, one or more instructions below the reconvergence point appear between the execution of the branch and the execution of the reconvergence point. The point marked RP in the control flow graph shown in Fig. 1C is an example of Rebound Reconvergence.

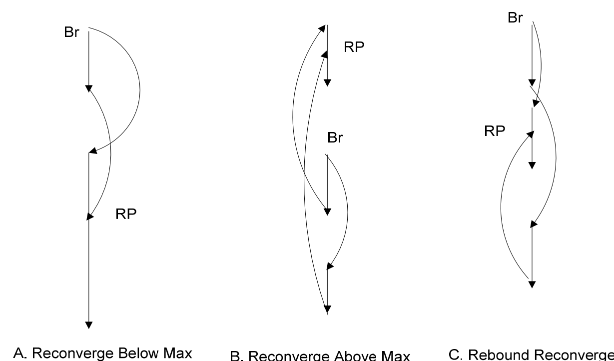

Fig. 1. The three dominant reconvergence categories

2.1 Training the CTW Reconvergence Predictor

We now briefly describe the training and prediction algorithm for the CTW predictor. The technique uses a reconvergence prediction table (RPT) that holds entries for various static branches. Each entry contains a potential reconvergence

[2] Below means a point in the program with larger program counter value.
[3] Above means a point in the program with smaller program counter value.

program counter (PC) for each of the three main convergence categories and a corresponding active bit which indicates that training is ongoing for the corresponding reconvergence potential.

When a branch is encountered, the three active bits are set and the training process proceeds as follows:

1. RBM training: the RBM potential is initialized to the PC just below the branch the first time a static branch executes. On subsequent encounters of the branch, the RBM potential is updated to a new PC, if the new PC is: (1) below the RBM potential in the RPT entry, and (2) is encountered after the branch executes but before the current RBM potential in the RPT entry executes.
2. RAM training: the RAM potential is initialized to 0 the first time a branch is executed. On subsequent encounters of the branch, the RAM potential is updated to a new PC, if the new PC is: (1) below the current RAM potential in the RPT and above the branch PC and (2) is encountered after the branch executes but before the current RAM potential in the RPT entry executes.
3. RR training: When the branch executes for the first time and every time the RBM potential changes, the RR potential in the RPT is set to the static instruction below the branch. On subsequent encounters of the branch, after the RBM potential is executed, the RR potential is updated to any PC observed that is below the branch and the current RR potential but above RBM potential. The RR potential becomes inactive whenever it is executed.

The above training for each category takes place when the active bit corresponding to the category is set, which is done every time the branch corresponding to the RPT entry executes. The active bit corresponding to a reconvergence category is cleared when the category potential is updated or the category potential is reached by the control flow. Training then becomes inactive until the same branch executes again, at which time another round of the training process starts.

2.2 Making Predictions with CTW Reconvergence Predictor

Since there are three reconvergence potential PCs in the RPT for each branch, additional three bits of information for each of the three categories are kept in the RPT entry to select the best candidate: ReachedFirst bit, ARTaken bit, and ARNTaken bit. Initially and whenever the corresponding reconvergence potential is updated, these bits are set. ReachedFirst bit, when set, indicates which of the three convergence potentials is reached first after the RPT entry branch executes. ARTaken bit, when set, indicates that the reconvergence potential is always reached when the RPT branch is taken. ARNTaken bit, when set, indicates that the reconvergence potential is always reached when the RPT entry branch is not taken.

Using the above state information, a reconvergence PC is predicted for a branch using the following algorithm: (1) Predict the reconvergence PC that has its reached first flag set, else (2) Predict some reconvergence PC that is always reached whether the branch is taken or not taken, else (3) Predict some reconvergence PC that is

always reached when the branch is either taken or always reached when the branch is not taken, else (4) Predict the RBM potential.[4]

Using the above CTW training and prediction algorithm, and defining a prediction as correct when the predicted dynamic reconvergence point is reached before the next statically determined reconvergence, [3] reports prediction accuracy larger than 99% on SpecInt benchmarks [13]. As we show in the next section, although this hit rate is quite high, the definition of control independence is loose and the reported number does not reflect a stricter, more accurate definition.

3 Limitation of CTW Control Independence Prediction Method

In this section we provide three examples for which the CTW control reconvergence prediction algorithm does not satisfy a strict definition of control independence. We assume that execution sequences in the following examples start with the first dynamic execution of branch Br, so training starts with the first instance of the given branch.

Consider the control flow graph in Fig. 2A which represents a simple loop with backward branch Br. In this example, ap is the RAM potential detected during the CTW training process, while bp is the RBM potential. Notice that bp is control independent of Br, since execution ultimately reaches this point when Br executes not-taken and exits the loop. On the other hand, ap is not a valid control independence point of Br. It is possible for the branch after the first loop iteration to execute not-taken, immediately exiting the loop without branching back to ap.

The CTW algorithm continuously predicts ap to be a dynamic control independence point of Br after a Br execution sequence of taken, taken, taken, not-taken, not-taken, taken. The reason is that on the first taken execution of Br, the algorithm initializes the predictor with bp and 0 as RBM and RAM potentials respectively with their ReachedFirst bits set. On the second taken execution of Br, RAM is set to ap. On the third execution of Br, ap is reached first, therefore clearing bp ReachedFirst bit. Afterwards, the potential selection algorithm predicts ap since it is the only reconvergence potential with the ReachedFirst bit set. TABLE I shows the prediction table state after each Br execution instance in this example.

[4] [3] Assigns highest priority to procedure return reconvergence. We do not consider return reconvergence here since it does not change our observations.

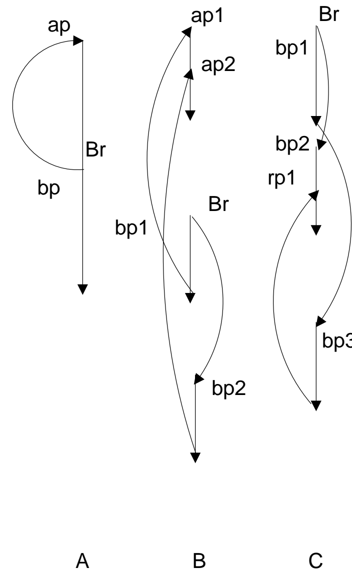

Fig. 2. Examples of control flow graphs

Notice that since the loop branch Br is executed taken a lot more than not-taken, and since ap is reached before the correct static independent control point bp, the prediction meets the definition of correctness in [3] most of the time, even though ap is not a truly control independent point.

TABLE I
CTW Prediction Table State for Example 2A

Br Outcome	T	T	T	NT	NT	T
RAM Value	0	ap	ap	ap	ap	ap
RBM Value	bp	bp	bp	bp	bp	bp
RAM ReachedFirst	1	1	1	1	1	1
RBM ReachedFirst	1	1	0	0	0	0

Consider now the more complex control flow graph example in Fig. 2B. In this example, ap1 and ap2 are two RAM potentials recorded in the predictor at different times during the training, while bp1 and bp2 are two RBM potentials. Br is the branch for which the predictor is trained. The correct control independence point that is always reached regardless of whether Br is taken or not taken is ap2. However, after a Br execution sequence of not-taken, not-taken, taken, taken, not-taken, not-taken, taken, the state in the CTW predictor would have ap2 and bp2 as the RAM and RBM

potentials with the ap2 ReachedFirst bit cleared and bp2 ReachedFirst bit set. Afterwards, the predictor will always give the incorrect prediction that bp2 is the control independence reconvergence point for Br. Again, this is counted as a correct prediction in [3] whenever Br is taken, since execution reaches bp2 first before reaching the correct static control independent point ap2. TABLE II shows the prediction table state after each Br execution instance in this example.

TABLE II
CTW Prediction Table State for Example 2B

Br Outcome	NT	NT	T	T	NT	NT	T
RAM Value	0	ap1	ap2	ap2	ap2	ap2	ap2
RBM Value	bp1	bp1	bp2	bp2	bp2	bp2	bp2
RAM ReachedFirst	1	1	1	0	0	0	0
RBM ReachedFirst	1	1	1	1	1	1	1

Finally, consider the control flow graph example in Fig. 2C. In this example, bp1, bp2 and bp3 are three RBM potentials recorded in the predictor at different times during the training. RR is assigned the PC of bp1. RR is not updated with the PC of rp1 since it becomes inactive after each execution of bp1. rp1 is the only control independent point for branch Br in the control flow graph.

After a Br execution sequence of not-taken, taken, not-taken, not-taken, not-taken, taken, not-taken, bp3 and bp1 are left in the predictor as the RBM and RR potentials respectively with bp3 ReachedFirst bit clear and bp1 ReachedFirst bit set. The CTW predictor then always predicts bp1 as the reconvergence point.

TABLE III shows the prediction table state after each Br execution instance in this example.

TABLE III
CTW Prediction Table State for Example 2C

Br Outcome	NT	T	NT	NT	NT	T	NT
RR Value	bp1	bp1	bp1	bp1	bp1	bp1	bp1
RBM Value	bp1	bp2	bp3	bp3	bp3	bp3	bp3
RR ReachedFirst	1	1	1	1	1	1	1
RBM ReachedFirst	1	1	1	0	0	0	0
RR Active bit	0	1	0	0	0	1	0

Even if RR is properly updated to the PC of rp1, for the above Br execution sequence, the CTW predictor will always predict bp3 as the reconvergence point instead of rp1.

TABLE IV shows the prediction table state after each Br execution instance in the above example.

TABLE IV
CTW Prediction Table State for Example 2C Assuming RR Set to rp1

Br Outcome	NT	T	NT	NT	NT	T	NT
RR Value	bp1	bp1	bp1	rp1	rp1	rp1	rp1
RBM Value	bp1	bp2	bp3	bp3	bp3	bp3	bp3
RR ReachedFirst	1	1	1	1	0	0	0
RBM ReachedFirst	1	1	1	1	1	1	1

As we have mentioned earlier, large tasks reduce the overhead of forking and committing tasks relative to the total execution time. However with large tasks, the performance penalty of control independence mispredictions is large, hence the need for highly accurate predictor. In the next section, we develop the CTW predictor further into a new algorithm that unlike CTW does not suffer from false positive mispredictions.

4 New Control Independence Prediction Scheme

In this new algorithm, we define the potential reconvergence points as defined in the CTW predictor. However, we eliminate CTW false positive predictions by properly training some of the RPT fields and providing an appropriate selection algorithm.

4.1 Training

RAM and RBM values are updated during training as in the CTW algorithm. For RR, only after an updated RR potential is executed, it becomes inactive. This avoids the scenario in TABLE III where the RR potential remains equal to its initial value and it is never updated. The training of the other RPT fields and the process of selecting which reconvergence potential to predict differ in our algorithm.

Always reach flags (ARTaken and ARNTaken): Initially and after the corresponding potential is updated, these bits are set. If the corresponding potential is not reached after the branch executes and before a second execution of the branch, one of these two bits is cleared, depending on whether the last branch execution was taken or not-taken. Before a new execution of the branch, we check if any potential has its active bit set. Having the active bit set indicates that the potential has not been neither executed nor updated after the last execution of the branch. For each of the found potentials if any, if the always reach flag corresponding to the outcome of the previous execution of the branch is set, the flag is cleared.

ReachedFirst bits: Initially and after each dynamic execution of a branch, the ReachedFirst flags of all three potentials are set to true in the branch RPT entry.

4.2 Making Predictions

The predicted reconvergence PC value is selected from among the potentials according to this algorithm:

1. Select a potential if after its execution, its ReachedFirst, ARTaken and ARNTaken flags are set. ReachedFirst flags of the other potentials are cleared when this selection is made.
2. Else, select RBM.

Whenever any of the potential is updated, no reconvergence PC is predicted.

4.3 Examples of Selecting Reconvergence Point with the New Algorithm

For the control flow graph in Fig. 2A, consider the Br execution sequence of taken, taken, taken, not-taken, not-taken, taken from section 3. The algorithm initializes the predictor after the first execution of Br. On the second taken execution of Br, RAM is set to ap. On the third execution of Br, ap is selected as control independence point of Br, RBM ReachedFirst and ARTaken are cleared. On the fourth execution of Br, RBM ReachedFirst is set and bp is predicted to be the control independence point of Br. On the fifth execution of Br, RAM ARNTaken is cleared. From now on, bp will always be selected as the control independence point of Br. TABLE V shows the state of the relevant entries of the prediction table after each Br execution instance in this example.

For the control flow graph example in Fig. 2B, consider the Br execution sequence of not-taken, not-taken, taken, taken, not-taken, not-taken, taken from section 3. The predictor is initialized after the first execution of Br. On the second execution of Br, RAM is set to ap1. On the third execution of Br, RAM and RBM are set to ap2 and bp2 respectively. On the fourth execution of Br, bp2 is selected as the reconvergence point and RAM ReachedFirst is cleared. On the fifth execution of Br, ap2 is selected as the reconvergence point and RBM ReachedFirst is cleared. On the sixth execution, RBM ARNTaken is cleared since bp2 was not encountered during the previous not-taken execution of Br. From now on, ap2 that satisfies the selection criterion 1 in subsection 4.2 above is always correctly predicted to be the control independence point of Br. TABLE VI shows the state of relevant entries in the prediction table after each Br execution instance in this example.

For the control flow graph example in Fig. 2C, after the Br execution sequence from section 3 of not-taken, taken, not-taken, not-taken, not-taken, taken, not-taken, rp1 satisfies the selection criterion 1 in subsection 4.2 above and is predicted to be the control independence point of Br. Afterwards, the predictor will always give the correct prediction rp1. TABLE VII shows the state of relevant entries in the prediction table after each Br execution instance in this example.

TABLE V
New Algorithm RPT State for Example 2A

Br Outcome	T	T	T	NT	NT	T
RAM Value	0	ap	ap	ap	ap	ap
RBM Value	bp	bp	bp	bp	bp	bp
RAM ReachedFirst	1	1	1	1	1	1
RBM ReachedFirst	1	1	0	1	1	1
RAM ARTaken	1	1	1	1	1	1
RAM ARNTaken	1	1	1	1	0	0
RBM ARTaken	1	1	0	0	0	0
RBM ARNTaken	1	1	1	1	1	1
Reconvergence Point	0	0	ap	bp	bp	bp

TABLE VI
New Algorithm RPT State for Example 2B

Br Outcome	NT	NT	T	T	NT	NT	T
RAM Value	0	ap1	ap 2	ap2	ap2	ap2	ap2
RBM Value	bp1	bp1	bp 2	bp2	bp2	bp2	bp2
RAM ReachedFirst	1	1	1	0	1	1	1
RBM ReachedFirst	1	1	1	1	0	0	0
RAM ARTaken	1	1	1	1	1	1	1
RAM ARNTaken	1	1	1	1	1	1	1
RBM ARTaken	1	1	1	1	1	1	1
RBM ARNTaken	1	1	1	1	1	0	0
Reconvergence Point	0	0	0	bp2	ap2	ap2	ap2

TABLE VII
New Algorithm RPT State for Example 2C

Br Outcome	NT	T	NT	NT	NT	T	NT
RR Value	bp1	bp1	bp1	rp1	rp1	rp1	rp1
RBM Value	bp1	bp2	bp3	bp3	bp3	bp3	bp3
RR ReachedFirst	1	1	1	1	0	1	1
RBM ReachedFirst	1	1	1	1	1	0	0
RR ARTaken	1	1	1	1	1	1	1
RR ARTaken	1	1	1	1	1	1	1
RBM ARTaken	1	1	1	1	1	1	0
RBM ARNTaken	1	1	1	1	1	1	1
Reconvergence Point	0	0	0	0	bp3	rp1	rp1

5 Constructing Large Tasks

The SpMT architecture we are developing includes a task dispatcher that assigns tasks to idle cores. We choose to spawn tasks in program order to simplify data communication due to the fact that two consecutive tasks execute on two adjacent cores. By dispatching tasks at control independent points, we ensure that the end of a given task is the beginning of the next sequentially dispatched task. Branches mispredictions are handled locally within a core without having any impact on other tasks. The instruction level parallelism that can be exploited is far beyond mispredicted branches. The size of the task should be sufficiently large to hide forking and committing overhead. Usually the number of instructions from a branch to its reconvergence point is less than the desired task size. We devise an algorithm that can be used to construct and dispatch large tasks.

We define the lowest RAM (LRAM) as the RAM with the lowest PC ever recorded for the branch. This means no instruction above the LRAM can execute between the execution of the branch and the execution of its LRAM.

We define a branch execution region as the set of static instructions that fall between the branch and its RBM when the RAM is not set to a valid value and between the LRAM and the RBM when the RAM has a valid value. Any instruction that falls within a branch execution region might be executed between the execution of the branch and its control independent reconvergence point. Fig. 3 shows the execution region of a branch. LRAM and RBM are respectively the first and last static instructions in the region.

A branch with a valid reconvergence point is called trained branch.

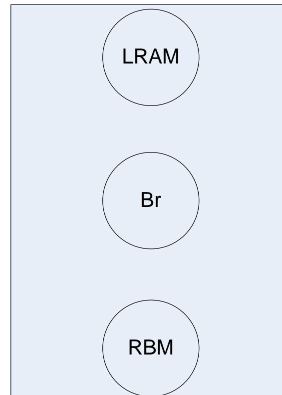

Fig. 3. Branch execution region.

Let Br1 be a trained branch with reconvergence point (RP) RP1. After the execution of RP1 we start looking for the first encounter of another trained branch. Let Br2 be the first trained branch executed after RP1 execution. Let RP2 be the predicted reconvergence point of Br2. If both Br2 and its reconvergence point RP2 do not fall within Br1 execution region, we construct a task composed of Br1 execution region, instructions executed between RP1 and Br2 executions and Br2 execution region. We call Br1 source branch (SB) and Br2 the target branch (TB) of Br1. We add a new field to each RPT entry that is used to point to the TB. If the estimated number of dynamic instructions within the constructed task is less than the desired task size, we repeat what we have done looking for the first trained branch encountered after RP2 execution and constructing a new larger task. We keep repeating this till we reach the desired task size. Let RPn be the reconvergence point of the last execution region added to the task. A speculative task can be spawned at Br1, where Br1 is the spawning instruction, an instruction that creates a new task when it is reached, and RPn is the spawned instruction, an instruction where the speculative task starts its execution. Suppose that the constructed task is scheduled to execute on core i, the speculative task can be spawned to execute on core (i+1) (mod m) starting at instruction RPn where m is the total number of cores.

We require that:

1) TB does not fall within SB execution region since our goal is to create a larger task. Any instruction within SB execution region might be executed between the execution of SB and its RP. This requirement is stricter than control independence, since any instruction that does not fall within the execution region of a branch is also control independent on the branch.
2) The RP of the TB does not fall within SB execution region. It has to be control independent on SB since it might be selected as a spawned instruction. We further require that it does not fall within SB execution region to avoid overlap between two adjacent tasks instructions since overlapping can increase inter-tasks dependences.

Fig. 4 illustrates a task constructed using our algorithm.

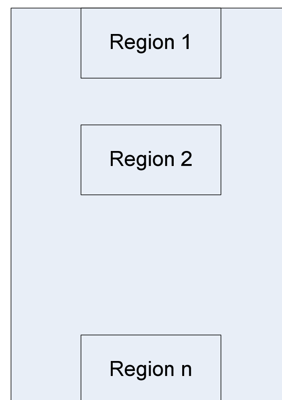

Fig. 4. Task constructed using the proposed algorithm.

4.1 Re-linking the Source Branch

Since the potentials are dynamically updated as the branch is executed, we check if the above two requirements are still satisfied each time one of the potentials of the SB is updated and we check if condition 2 above is still satisfied each time the RP of the TB is updated. If one of the above requirements is violated, we clear the link pointer and start looking for another target branch.

Now that we have described our new algorithms, we discuss our simulation methodology and results.

6 Simulation Methodology and Results

We use SPEC CPU 2000 integer benchmarks [13] as our target benchmark suite. These benchmarks exhibit a wide variety of complex control flow behavior making them suitable for testing our algorithms.

We have modeled the proposed algorithms using PTLsim [16][17]. PTLsim is a cycle accurate microprocessor simulator and virtual machine for x86 and x86-64 instruction sets. It simulates x86 codes after converting complex instructions into RISC-like micro-ops (uops), a technique used in Intel and AMD processors.

6.1 Checking the Accuracy of the Predictor

We have implemented the following method to check for the accuracy of the proposed control independence predictor. We use the definitions of the three reconvergence categories from section 2. Whenever the predictor makes a prediction for a branch, the simulator monitors the program counter values of the subsequent retired instructions. If an instruction with a program counter that violates the definition of the predicted reconvergence category is encountered before the predicted reconvergence instruction, the prediction is counted as a miss. Otherwise, if all instructions between the branch and the next encounter of the predicted reconvergence instruction satisfy the definition of the predicted reconvergence category, the prediction is a hit.

Our simulations indicate that after sufficient period of initial training to warm up the predictor and to capture all the control flow graph information into the prediction table state, the predictor correctly predicts control independent points with 100% accuracy.

7 Conclusion

We have presented in this paper a dynamic control independence predictor that completely accounts for the control flow graph structure and identifies accurately control independent points. This predictor can be used in any processor architecture that exploits control independence. Our predictor does not require any support from the compiler or modification to the underlying ISA. We have also described an algorithm that can be used to construct large tasks.

References

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture,* November 1998.

[2] C. Cher and T. Vijaykumar. Skipper: A Microarchitecture for Exploiting Control Flow Independence. In *Proceedings of the 34th International Symposium on Microarchitecture*, November 2001.

[3] J. D. Collins, D. M. Tullsen and H. Wang. Control Flow Optimizations Via Dynamic Reconvergence Prediction. In *Proceedings of the 37th Annual International Symposium on Microarchitecture,* December 2004.

[4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, January 1989.

[5] M. Franklin and G. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[6] A. Gandhi, H. Akkary, and S. Srinivasan. Reducing Branch Misprediction Penalty via Selective Branch Recovery. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture,* February 2004.

[7] P. Marcuello and A. Gonzalez. Thread-spawning schemes for multithreaded architectures. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, February 2002.

[8] P. Marcuello, A. González, and J. Tubella. Speculative Multithreaded Processors. In *Proceedings of International Conference on Supercomputing'98*, July 1998.

[9] K. Olukotun, L. Hammond, and M. Willey. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. In *Proceedings of the 13th International Conference on Supercomputing*, June 1999.

[10] E. Rotenberg, Q. Jacobson, and J. Smith. A Study of Control Independence in Superscalar Processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture,* January 1999.

[11] J. Smith and G. Sohi. The Microarchitecture of Superscalar Processors. In *Proceedings of the IEEE,* (83)12, December 1995.

[12] G. Sohi, S. Breach and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[13] Standard Performance Evaluation Corporation. SPEC benchmarks. http://www.spec.org/

[14] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, January 1998.

[15] J.-Y. Tsai, Z. Jiang, E. Ness, P.-C. Yew. Performance Study of a Concurrent Multithreaded Processor. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture,* February 1998.

[16] X86 Cycle Accurate Processor Simulation Design Infrastructure. http://www.ptlsim.org/

[17]M. T. Yourst. Ptlsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator, In *Proceedings of 2007 International Symposium on Performance Analysis of Systems and Software (ISPASS2007),* pp. 23-34, January 2007.

Received: June, 2009