A Novel Approach to Control Reconvergence Prediction

Walid J. Ghandour*, Nadine J. Ghandour†
* Fahad Bin Sultan University, Tabuk, Saudi Arabia
Email: ghandour@utexas.edu
† Lebanese University, Beirut, Lebanon
Email: nadin.ghandour@ul.edu.lb

***Abstract*—Instruction Level Parallelism (ILP) increases the overlap between instructions to enhance performance. Control dependences, introduced by conditional branch instructions, present a limiting factor on ILP. A reconvergence point is an instruction at which the control flow eventually reconverges regardless of the outcome or target of the current control flow instruction. Execution following the reconvergence point is certain. Instructions after the reconvergence point are fetched irrespective of the branch's outcome since they are control-independent.**

This article presents three algorithms. The first algorithm predicts future control reconvergence points in the dynamic instruction stream. The second algorithm predicts control-independent data-independent instructions. The third algorithm is used to construct large tasks suitable for speculative multithreading architectures (SpMT).

We compare the proposed control reconvergence prediction technique with the state of art prediction techniques and show that it achieves 100% accuracy using SPEC CPU 2006 integer benchmarks. We provide a qualitative comparison of the proposed control-independent data-independent instructions prediction technique with existing techniques. We show the potential of the proposed technique to construct large tasks for SpMT architectures through using it in a SpMT architecture that we are developing where it outperforms the fork-on-call and multiple-proc-fork techniques.

***Keywords*—computer architecture, control flow reconvergence, control independence, data independence, dynamic slicing, instruction level parallelism, speculative multithreading.**

I. Introduction

The availability of large number of transistors on chip has enabled the computer industry to move to many-core architectures which have become mainstream for both general-purpose and embedded computing [1], [2]. As the number of cores increases, communication latency increases and communication overhead may dominate execution time [1], [2].

The majority of programmers are educated to program in the sequential programing style. Current education approach is still more oriented toward sequential programing. Parallel programing is more difficult, time consuming and error prone than sequential programing. Sequential programing is expected to remain dominant for the foreseeable future. Also, future parallel applications are expected to exhibit large sequential sections and a sequential control. According to Amdahl's law, the execution time of a parallel application cannot be reduced below the execution time of its residual sequential part. Therefore, processor chips are required to achieve high performance on both sequential codes and sequential sections of parallel applications.

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 [3]. Since for each control flow instruction, regardless of the path taken, execution will eventually converge to a control independent block of instructions [4], 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 (RP), since the control flow eventually reconverges at this point, independent of the execution of the current control flow instruction. Speculative Multithreading (SpMT) architectures [5], [6], [7], [8] 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. If the control flow predictor in a SpMT processor predicts tasks correctly at future control reconvergence 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

978-1-4673-7166-7/15/$31.00

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 article has resulted from this need for highly accurate control independence prediction.

Program slicing is an approach that determines the set of statements on which the value of a given variable depends. Dynamic slicing considers runtime information when selecting the set of statements that affects a specific program variable. In this article we show the potential of using dynamic slicing to predict control reconvergence points and construct large tasks for SpMT architectures.

A. Article Contributions

This article makes the following contributions:

- Proposes a dynamic slicing directed control reconvergence prediction technique.
- Introduces a dynamic slicing directed algorithm that predicts control-independent data-independent instructions.
- Presents an algorithm for constructing large tasks for SpMT architectures. This gives the SpMT architecture the advantage of small task dispatch and data communication overhead relative to the total task execution time.
- Shows the potential of the proposed techniques.

B. Article Overview

The rest of this article is organized as follows. Section II provides background information and reviews related work. Section III presents the methodology we follow. Section IV introduces dynamic slicing directed control reconvergence prediction technique. Section V describes a method to predict control-independent data-independent instructions. Section VI extends control reconvergence prediction to make use of it in SpMT architecture. Section VII shows the potential of the proposed techniques. Finally, we conclude in Section VIII.

II. Background and Relevant Work

A. Control Independence Prediction Overview

Control independence [4] 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 [9], 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 account for rarely or never executed paths, since a compiler lacks dynamic information.

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) [10] 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 reconvergence 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 [11] and Selective Branch Recovery [12]. 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 is a more aggressive approach for exploiting dynamic control independence [13]. 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 [14] proposes a general technique that does not depend on the compiler or on heuristics to predict various forms of control independence. The control independence predictor identifies three major categories of control flow reconvergence. It involves training and prediction algorithms. The technique uses a reconvergence prediction table that holds entries for various static branches. The training for each branch is activated every time the branch executes. Training continues till either the potential reconvergence point is updated or the selected potential reconvergence 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. Each branch has a potential reconvergence point for each of the three convergence categories identified. The prediction algorithm selects one of them as the reconvergence point of the instruction. We will refer in the rest of the article to this control independence predictor as CTW predictor, after the authors' initials.

B. Speculative Multithreaded Processors

Speculative multithreaded processor (SpMT) is an architecture that try to enhance the performance of single threaded applications by partitioning an application into several threads or tasks, each executing on a different context or thread unit. SpMT provides an effective large instruction window built up of nonadjacent smaller windows created using speculation on highly predictable branches.

Different approaches can be used to spawn tasks. These approaches can slightly change from an architecture to another. Task spawning policy selects the portions of the code to be executed by speculative tasks and determines when these tasks are forked. In several SpMT architectures such

as Multiscalar [15], Single-program speculative multithreading (SPSM) [16] and Superthreaded [17] architectures, speculative tasks are generated using the compiler. In other architectures, such as Dynamic Multithreaded Processor [10], Speculative Multithreaded Processor [6] and Clustered Speculative Multithreaded Processor [18], hardware techniques are used to partition the program at run-time. Spawning point is the instruction at which a new task is created. Control quasi-independent point or speculation point is the instruction where the speculative task starts execution.

C. Information Flow Analysis

A statement *t* is data dependent on a statement *s* if *t* uses a variable that is defined in *s*. A statement *t* is control dependent on a statement *s* if *s* decides via branches it controls whether *t* is executed or not [19]. Information flows from a source object *x* to a target object *y* whenever information stored in *x* is passed directly or indirectly to object *y* [20]. A given program statement designates a specific flow *f* if executing the statement could result in the flow *f*. Flows can be either direct or indirect.

Direct information flow is due to the propagation of information as a result of data dependence. Source variable is directly involved in the computation of the target variable value. The occurrence of a direct flow is independent of the value of its source object. Indirect information flow is due to control dependence. Source object indirectly affects the value of the target object.

Program slicing is an approach that determines the set of statements on which the value of a given variable depends. It was originally introduced by Weiser [21] who proposed the first static slicing algorithm to help debugging programs. Afterward, Korel and Laski [22] proposed dynamic slicing that considers runtime information when selecting the set of statements that affects a specific program variable. The proposed algorithms use *backward* analysis, which requires the a priori availability of the execution trace. Forward computing dynamic slicing, initially introduced by Korel and Yalamanchili [23], and information flow analysis algorithms, which we use in this article, do not need the a priori availability of the execution trace.

Now that we have reviewed background and previous relevant work, we discuss our methodology.

III. Methodology

Pin [24] is a dynamic binary instrumentation tool. Instrumentation is a technique that inserts code into a program to collect run-time information. *Pin* does not need the source code, and does not require recompilation and post-linking. It provides rich APIs that allow the user to write instrumentation tools called pintools in C, C++ or assembly. It is supported on Linux and Windows for x86, x86-64 and Itanium architectures. It can instrument real-life applications such as database and web browsers, and multithreaded applications. We use *Pin* to implement a dynamic slicing tool for x86 binaries. The tool tracks control and data dependences between assembly instructions. We employ the tool to study dynamic slicing directed control reconvergence prediction and to identify control-independent data-independent instructions.

To measure the potential performance of the proposed techniques, we have modeled our multi-core architecture using PTLsim [25]. 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 [26]. We compare the performance of our architecture to that of a large 4-wide out-of-order superscalar processor that approximates current state-of-the-art processors. We size all buffers of our architecture to one fourth the size of the out-of-order core buffers.

We conduct our study using SPEC CPU 2006 integer benchmarks [27]. These benchmarks exhibit a wide variety of complex control flow behavior making them suitable for testing our algorithms.

We describe in the following section the proposed control reconvergence prediction technique.

IV. Dynamic Slicing Directed Control Reconvergence Prediction

A control reconvergence point is an instruction at which the program execution paths will converge regardless of the outcome or target of the current branch. Instructions between a branch and its reconvergent point are control-dependent (CD) on the branch. Instructions after the reconvergent point are fetched irrespective of the branch's outcome, they are control-independent (CI). Some CI instructions are influenced by the branch through register or memory data dependences, they are control-independent data-dependent (CIDD). Control independent instructions that are not affected by the branch in any way are called control-independent data-independent (CIDI) instructions.

Conventional superscalar processors flush all instructions following a mispredicted branch and restart by fetching instructions from the right path. CI instructions need not to be *refetched*, and CIDI instructions need not to be *reexecuted*. Control independence can also be proactively used before a conditional branch misprediction occurs. When the fetch engine encounters a difficult to predict branch, it can start fetching control independent instructions instead of predicting the outcome of the branch.

Based on control flow following branch execution, reconvergence points can be either below the branch instruction, or above the branch instruction. Reconvergence points below the branch instruction exhibits 93% of static branches [14]. We divide branch instructions into two categories: reconvergence below maximum and reconvergence above maximum. Figure 1 illustrates these two categories.

- Reconverge Below Maximum (RBM): This category refers to a branch for which the control independent reconvergence point in the program is below the branch. Moreover, no instruction below the reconvergence point can appear between the execution of the branch and the execution of the reconvergence point.
- Reconverge Above Maximum (RAM): This category refers to a branch for which the control independent reconvergence point is above 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.

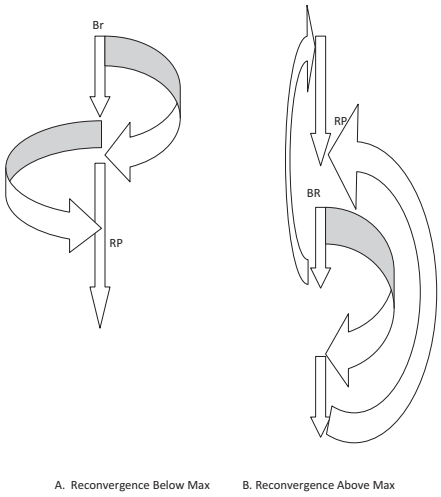

Fig. 1. The two reconvergence categories.

In the following section we describe our proposed control reconvergence prediction technique.

A. Control Reconvergence Prediction

We provide a profiling based dynamic Slicing Directed control Reconvergence prediction technique (SDR). It only requires support to pass hints to the hardware which can be either implicit or explicit [28], [29].

Following are the steps used to identify a control reconvergence point of a given conditional branch:

1) Identify the immediate post-dominator[1], *ipdom*, of each executed branch instruction.
2) When the *ipdom* is below the branch instruction, e.g. its instruction pointer (IP) is greater than that of the branch, the reconvergence point should fall between the branch and its *ipdom*. The control reconvergence point region is $]IP_{branch}, IP_{ipdom}]$.
3) When the *ipdom* is above the branch instruction, e.g. its IP is smaller than that of the branch, the reconvergence point should fall above or at the *ipdom*, i.e. its IP is less than or equal to that of the *ipdom*.
4) If the branch instruction is both taken and non-taken during the program execution, select instructions in the control reconvergence point region, defined in steps 2 or 3 above, that are encountered when the branch is both taken and non-taken. This can be identified by the existence of flow from the branch to the target instruction when the branch is both taken and non-taken.
5) If the branch instruction is only taken or non-taken during the program execution, select instructions in the control reconvergence point region that are encountered when the branch is executed.
6) When more than one instruction are selected in steps 4 or 5 above, we select the instruction at the largest distance from the *ipdom*, i.e. $abs(IP_{ipdom} - IP_{reconvergence})$ is the largest among the selected instructions. The selected instruction is predicted as the control reconvergence point of the branch instruction.
7) If no instruction is selected, the branch *ipdom* is selected as its control reconvergence point.

Algorithm 1 illustrates the dynamic slicing directed control reconvergence prediction technique. Following are the definitions of the symbols used in the algorithm: (i) $isCondBr(ins)$: Returns true if ins is a conditional branch instruction. (ii) $IP_{ipdom(ins)}$: Instruction pointer of the ipdom of instruction ins. (iii) $ReconvSet(Br)$: Set of instructions that are always reached after the execution of conditional branch Br. (iv) $ReconvPt(Br)$: Reconvergence point of Br.

B. Comparison of Different Prediction Techniques

We compare in this section the accuracy of SDR with that of the existing control reconvergence prediction techniques.

DMT [10] only predicts reconvergence points for loop backward branches and procedure calls. We extend it in a way that forward and indirect branches are predicted to converge at the reconvergence point of the innermost loop containing them, or at the reconvergence point of their corresponding function in case no loop contains them.

All DMT reconvergence predictions at function return are considered correct. For the remaining DMT predictions, CTW, SDR and Skipper, a reconvergence point is correctly predicted if it is encountered before the reconvergence point selected by the static scheme.

Figure 2 shows the accuracy from left to right of the following prediction techniques: static scheme, Skipper, DMT, CTW and SDR. The height of each bar in Figure 2 indicates the prediction accuracy of the corresponding scheme. We define prediction accuracy as the percentage of correctly predicted reconvergence points out of the total prediction attempts. Each bar is divided according to the dynamic number of executed instructions between the execution of the control instruction and the execution of its correctly predicted control reconvergence point. We call this number the dynamic distance between the control instruction and its reconvergence point.

SDR achieves 100% prediction accuracy on average for all benchmarks. This results is expected, since, SDR always selects a reconvergence point that will be encountered before the encounter of the ipdom of the control instruction, in case the ipdom itself is not selected as explained in Algorithm 1. For SDR, 62% of the predicted control reconvergence points are within the next 16 dynamic instructions, 74.18 % are within the next 32 dynamic instructions, and 82.55% are within the next 64 dynamic instructions. In contrast, for DMT 45.63% of the correctly predicted control reconvergence points are within more than 256 dynamic instructions of the corresponding control instruction.

The static approaches accounts for rarely or never executed paths. SDR is a dynamic technique that does not require complex hardware support and training such as those required for CTW.

[1]In control flow graphs, a node u is post-dominated by node v, iff all u - EXIT paths contain v. The immediate post-dominator is the post-dominator that appears first on any u - EXIT path.

Algorithm 1 Control Reconvergence Prediction Algorithm

```
1: if (isCondBr(ins)) then
2:    if (IP_ipdom(ins) > IP_ins) then
3:       Reconvergence point region of ins = ]IP_Br, IP_ipdom]
4:    else if (IP_ipdom(ins) < IP_ins) then
5:       Reconvergence point region of ins = ]0, IP_ipdom]
6:    end if
7: else if (Br is both taken and non − taken)
                  && (IP_ins ∈ ReconvRegion_Br) && (ins encountered when Br is both taken and non − taken)
   then
8:    ReconvSet(Br).insert(ins)
9: else if (Br is either taken or non−taken) && (IP_ins ∈ ReconvRegion_Br) && (ins encountered when Br is executed)
   then
10:   ReconvSet(Br).insert(ins)
11: end if
12: if (ReconvSet(Br).isEmpty()) then
13:   ReconvPt(Br) = ipdom(Br)
14: else if (ReconvSet(Br).size() == 1) then
15:   ReconvPt(Br) = ReconvSet(Br).get()
16: else
17:   ReconvPt(Br) = ReconvSet(Br).getFartherFromIpdom()
18: end if
```

Control reconvergence point prediction is done in software using dynamic slicing instrumentation and profiling, and propagated to the hardware using special compiler added hints. Hashed IP of the predicted reconvergence point, encoded in the branch instruction, is used to communicate compiler analysis to the architecture. Several processors instruction set architectures (ISA) already support hint bits. As an example, Itanium ISA [28] contains hint bits for branch prediction and memory locality. Researchers have proposed the usage of hint bits for value prediction [30]. Implicit hint is an approach that encodes hints in the registers names without changing the ISA [29].

In the following section, we describe our approach to identify control-independent data-independent instructions.

V. Control-Independent Data-Independent Instructions Prediction

Following are the steps used to identify control-independent data-independent (CIDI) instructions. These are instructions that are not affected by the branch in any way.

Let RP be the reconvergence point of a given conditional branch Br.

- If RP is below Br, i.e. $IP(RP) > IP(Br)$:
  - All instructions that fall in the control independent block of Br, and that don't have data dependence on any instruction that falls between $[Br, RP[$, are CIDI on Br.
- If RP is above Br, i.e. $IP(RP) < IP(Br)$:
  - Let LIP be the Lowest IP ever encountered between the execution of the branch and its RP; No instruction above the LIP, i.e. its IP is smaller than LIP, can execute between the execution of the branch and the execution of its LIP.
  - Let HIP be the Highest IP ever encountered between the execution of the branch and its RP; HIP is greater than $IP(Br)$.
  - All instructions that fall in the control independent block of Br, and that don't depend on any instruction that falls in $[LIP, RP[\cup [Br, HIP[$, are CIDI on Br.

Algorithm 2 illustrates the control-independent data-independent instructions prediction technique. Following are the definitions of the symbols used in the algorithm: (i) $IndependBlock_{Br}$: Control independence block of branch Br. (ii) $DependSet_{ins}$: Dependence set of instruction ins. This set contains the instructions on which ins is data dependent.

Algorithm 2 Control-Independent Data-Independent Instructions Prediction Algorithm

```
1: if (IP_RP > IP_Br) then
2:    if (ins ∈ IndependBlock_Br) then
3:       if (DependSet_ins ∩ [Br, RP[= φ) then
4:          ins is a CIDI
5:       end if
6:    end if
7: else
8:    if (ins ∈ IndependBlock_Br) then
9:       if (DependSet_ins ∩ ([LIP, RP[ ∪ [Br, HIP[) = φ)
         then
10:         ins is a CIDI
11:      end if
12:   end if
13: end if
```

A. Qualitative Comparisons With Related Work

Several proposal attempts to exploit control independence in superscalar [11], [31], [12], [4], [32], [33] and speculative

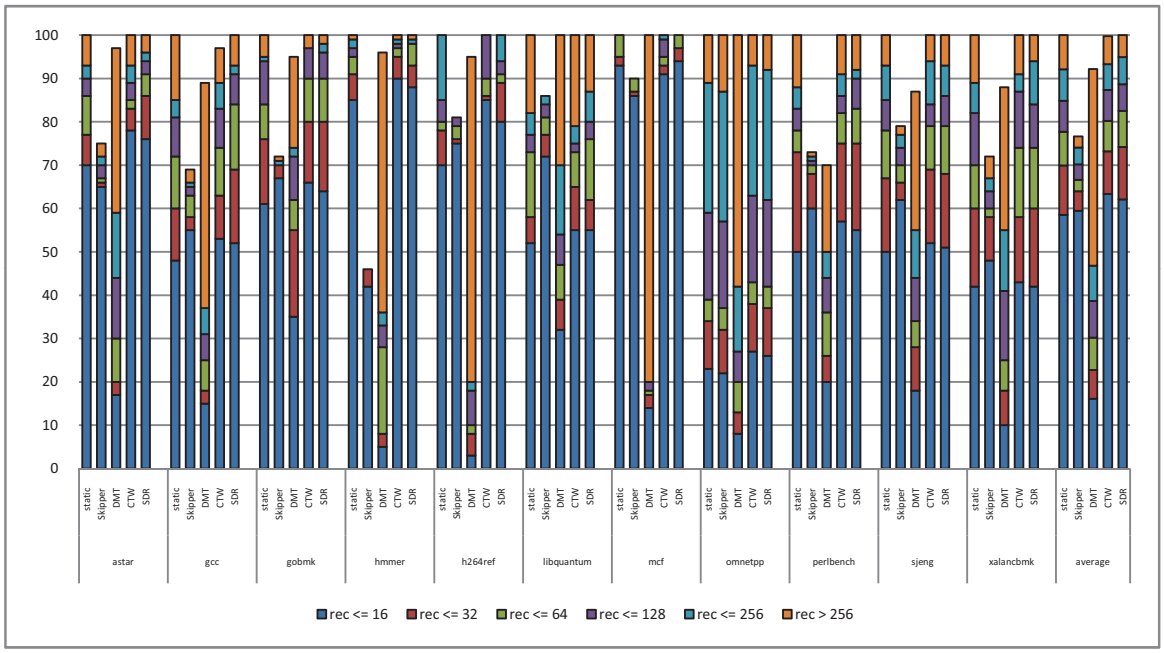

Fig. 2. Comparison of reconvergence prediction rates.

multithreaded processors [10], [7]. These approaches are hardware based. Three steps are mainly used to exploit control independence: i) insertion/removal of control dependent instructions, ii) re-renaming of control independent data dependent isntructions and iii) re-execution of control independent data dependent instructions.

To accomplish these steps, [4] and [32] need complex linked-list ROB management, [33] degrades performance using full flushing, [11] underutilizes ROB padding. [10] and [7] target speculative multithreaded application. Transparent Control Independence (TCI) [34] uses a coarse-grain checkpoint-based retirement strategy. It also requires the addition of several hardware components and lot of architecture modifications.

The approach provided in this article is profiling based. It collects the required information dynamically, and identify CIDI and CIDD instructions. The required information is passed to the processor via compiler support. A poisoning bit can be added to control independent (CI) instructions to indicate if they are CIDI or CIDD. CIDI instructions are issued and CIDD instructions wait till the branch is resolved.

In the following section, we present our approach to construct large tasks for SpMT architectures.

VI. Constructing Large Tasks

Speculative Multithreading (SpMT) architectures [5], [6], [7], [8] 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.

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 reconvergence 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 in this article an algorithm that can be used to construct and dispatch large tasks.

Based on control flow following branch execution, reconvergence points can be either below the branch instruction, or above the branch instruction. We classified branch instructions according to their reconvergence points in Section IV in two categories: Reconverge Below Maximum (RBM) and Reconverge Above Maximum (RAM).

We define a branch execution region as the set of instructions that might be executed between the execution of the branch and its control independent reconvergence point. For RBM branch instructions, the execution region is the set of static instructions that fall between the branch and its reconvergence point. For RAM branch instructions, the execution region is the set of static instructions that fall between its corresponding LIP and HIP instructions[2].

Let Br_1 be a branch with reconvergence point RP_1. After the execution of RP_1 we start looking for the first encounter of another branch. Let Br_2 be the first branch executed after RP_1 execution. Let RP_2 be the predicted reconvergence point of Br_2. If both Br_2 and its reconvergence point RP_2 do not

[2]We define LIP and HIP in Section V.

Baseline Core	15 stage, 4-wide, 160-entry ROB
Proposed Core	12 stage, 2-wide deocde, 1-wide issue
L1 Data Cache	Superscalar: 64KB, 8-way, 3 cycles, 64-byte line Proposed core: 16KB , 4-way, 2 cycles, 32-byte line
L1 Instruction Cache	Superscalar: 64 KB, 8-way, 3 cycles, 64-byte line Proposed core: 16KB, 4-way, 2 cycles, 32-byte line
L2 Cache	256 KB, 8-way, 12 cycles, 64-byte line
L3 Cache	1MB, 16-way, 40 cycles, 64-byte line
L3 to Memory Latency	150 cycles (assuming integrated DRAM controller)
Branch Predictor	Combined bimodal-gshare 64K Meta, 64K bimodal, 64K gshare 4K BTB, 4K indirect branch predictor

TABLE I. PARAMETERS FOR OUR ARCHITECTURAL CONFIGURATION.

fall within Br_1 execution region, we construct a task composed of Br_1 execution region, instructions executed between RP_1 and Br_2 executions, and Br_2 execution region. We call Br_1 source branch (SB) and Br_2 the target branch (TB) of Br_1.

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 branch encountered after RP_2 execution and constructing a new larger task. We keep repeating this till we reach the desired task size. Let RP_n be the reconvergence point of the last execution region added to the task. A speculative task can be spawned at Br_1, where Br_1 is the spawning instruction, an instruction that creates a new task when it is reached, and RP_n is the spawned instruction, an instruction where the speculative task starts its execution.

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.
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.

The pairs of spawning and spawned instructions are determined in software and propagated to the hardware, where a small hash table is used to store this information.

VII. EXPERIMENTAL RESULTS

The architecture we are developing combines SpMT with checkpoint recovery [35], [36], [37], [38] and continual flow pipeline [39]. It consists of a set of cores connected in a ring architecture. Tasks are spawned at future control reconvergence points in the program. This ensures that regardless of branch instructions outcomes within a task, a task will most probably join the next spawned task. Tasks are spawned in program order to ensure that two consecutive tasks run on two adjacent cores in the ring and to simplify data communication between tasks.

The configuration parameters for our simulations are given in Table I.

Figure 3 presents the performance of our proposed architecture relative to the baseline core for three forking methods. The first two methods fork tasks at future procedure continuation (i.e., return target) points. The fork-on-call method forks tasks

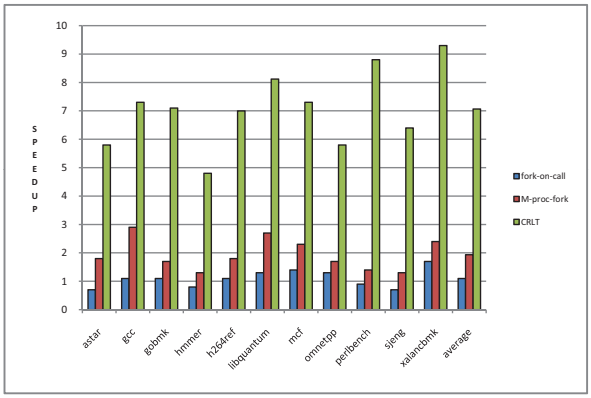

Fig. 3. Speedup over baseline of our proposed architecture that employs four cores using three distinct forking methods.

at the return target of the immediate procedure. Multiple-proc-fork forks tasks at a future procedure continuation after skipping a fixed number of procedures. The number of procedures to skip can be selected dynamically. This method provides better load balance than the fork-on-call method. In this article, we set the number of procedures to skip to four based on the observed performance of our benchmarks. The third method, Control Reconvergence directed Large Tasks technique (CRLT), uses the techniques presented in Sections IV-A and VI to construct tasks.

We observe that the fork-on-call method suffers from wide variation in task size, that ranges from less than hundred instructions per task and up to several thousands of instructions. This significant variation in task size induces major load imbalance. Cores that execute small tasks terminate quickly and then remain idle for a long time waiting for the preceding cores that are executing large tasks to finish. We also notice that several forked procedures are very small, less than thirty instructions. The dispatch and commit overhead is too high to benefit overall performance.

For our newly introduced task construction mechanism described in Section VI, we set the desired range of task size between 500 and 2000 instructions. The intention is to spawn tasks with sizes that are neither too large, nor too small.

Constructing tasks using the approach presented in Sections IV-A and VI, achieves 7.06%, 5.13% and 5.07 % speedup over baseline, fork-on-call and multiple-proc-fork respectively.

VIII. CONCLUSION

We presented in this article a dynamic control reconvergence predictor that can be used in any processor architecture that exploits control independence. We also described an approach to identify control-independent data-independent instructions. In addition, we introduced an algorithm that can be used to construct large tasks for speculative multithreading architectures. We applied the proposed techniques to a novel speculative multithreading architecture and showed that it outperforms the fork-on-call and multiple-proc-fork techniques.

REFERENCES

[1] J.D. Owens, W.J. Dally, R. Ho, D.N. Jayasimha, S.W. Keckler and L. Peh, "Research challenges for on-chip interconnection networks," *IEEE Micro*, vol. 27(5), pp. 96–108, September-October 2007.

[2] R.P. Martin, A.M. Vahdat, D.E. Culler and T.E. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," *SIGARCH Computer Architecture News*, vol. 25(2), pp. 85–97, 1997.

[3] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83(12), pp. 1609–1624, August 1995.

[4] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of control independence in superscalar processors," *HPCA'5*, January 1999.

[5] M. Franklin and G. Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," *ISCA*, pp. 58–67, May 1992.

[6] P. Marcuello, A. Gonzlez and J. Tubella, "Speculative multithreaded processors," *International Conference on Supercomputing'98*, July 1998.

[7] G. S. Sohi, S. E. Breach and T.N. Vijaykumar, "Multiscalar processors," *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pp. 414–425, June 1995.

[8] J. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," *HPCA'4*, January 1998.

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "An efficient method of computing static single assignment form," *Symposium on on Principles of Programming Languages*, January 1989.

[10] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," *Proceedings of the 31st International Symposium on Microarchitecture*, November 1998.

[11] C. Cher and T. Vijaykumar, "Skipper: A microarchitecture for exploiting controlflow independence," *In Proceedings of the 34th International Symposium on Microarchitecture*, November 2001.

[12] 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.

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

[14] J. D. Collins, D. M. Tullsen and H. Wang, "Control flow optimization via dynamic reconvergence prediction," *37th International Symposium on Microarchitecture*, December 2004.

[15] M. Franklin, "The multiscalar architecture," Ph.D. dissertation, University of Wisconsin, November 1993.

[16] P.K. Dubey, K. OBrien, K.M. OBrien and C. Barton, "Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading," *Proc. of the Int. Conf on Parallel Architectures and Compilation Techniques*, pp. 109–121, 1995.

[17] J.Y. Tsai and P-C. Yew, "The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation," *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 35–46, 1996.

[18] P. Marcuello and A. Gonzalez, "Clustered speculative multithreaded processors," *Proc. of the 13th Int. Conf. on Supercomputing*, pp. 365–372, 1999.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, July 1987.

[20] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communication of the ACM*, vol. 20(7), pp. 504–513, 1977.

[21] M. Weiser, "Program slicing," ser. ICSE '81, 1981, pp. 439–449.

[22] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, pp. 155–163, October 1988.

[23] B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices, issta," pp. 66–79, 1994.

[24] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," ser. PLDI '05, 2005.

[25] 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, 2007.

[26] D. B. Papworth, "Tuning the pentium pro microarchitecture," *IEEE MICRO*, vol. 16(2), April 1996.

[27] "The Standard Performance Evaluation Corporation (SPEC). The SPEC Benchmark Suite," http://www.spec.org.

[28] "Intel itanium architecture software developers manual," January 2006, document Number: 245319-005.

[29] H. Vandierendonck and K. D. Bosschere, "Implicit hints: Embedding hint bits in programs without isa changes," *ICCD*, pp. 364–369, Oct. 2010.

[30] Q. Zhao and D. J. Lilja, "Static classification of value predictability using compiler hints," *IEEE Transactions on Computers*, vol. 53, pp. 929–944, Aug 2004.

[31] Y. Chou, J. Fung, and J. P. Shen, "Reducing branch misprediction penalties via dynamic control independence detection," in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS '99, 1999, pp. 109–118.

[32] E. Rotenberg and J. Smith, "Control independence in trace processors," in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32, 1999, pp. 4–15.

[33] A. Sodani and G. Sohi, "Dynamic instruction reuse," *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, June 1997.

[34] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg and H. Akkary, "Transparent Control Independence (TCI)," *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[35] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-oforder execution machines," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, June 1987.

[36] K. Yeager, "The mips r10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996.

[37] D. Leibholz and R. Razdan, "The alpha 21264: A 500 mhz out-of-order execution microprocessor," *In Proceedings of the 42nd IEEE Computer Society International Conference*, February 1997.

[38] H. Akkary, R. Rajwar and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[39] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi and M. Upton, "Continual flow pipelines," *ISCA-11*, October 2004.