Probabilistic Points-to Analysis for Optimizing Speculative Multithreading Architecture

Peng-Sheng Chen †
pschen@pllab.cs.nthu.edu.tw
†Department of Computer Science
National Tsing Hua University
Hsinchu 300
Taiwan

Ming-Yu Hung†
myhung@pllab.cs.nthu.edu.tw
‡Department of Computer Science
National Taiwan Ocean University
Keelung 202
Taiwan

Yuan-Shin Hwang†
shin@cs.ntou.edu.tw
†Department of Computer Science
National Taiwan Ocean University
Keelung 202
Taiwan

Roy Dz-Ching Ju†
roy.ju@intel.com
§Microprocessor Research Lab.
Intel Corporation
Santa Clara, CA 95052
U.S.A.

Jeng Kuen Lee‡
jklee@cs.nthu.edu.tw

ABSTRACT

Speculative multithreading (SpMT) architecture can exploit thread-level parallelism that cannot be identified statically. Speedup can be obtained by speculatively executing threads in parallel that are extracted from a sequential program. However, performance degradation might happen if the threads are highly dependent, since a recovery mechanism will be activated when a speculative thread executes incorrectly and such a recovery action usually incurs a very high penalty. Therefore, it is essential for SpMT to quantify the degree of dependences and to turn off speculation if the degree of dependences passes certain thresholds. This paper presents a technique that quantitatively computes dependences between loop iterations and such information can be used to determine if loop iterations can be executed in parallel by speculative threads. This technique can be broken into two steps. First probabilistic points-to analysis is performed to estimate the probabilities of points-to relationships in case there are pointer references in programs, and then the degree of dependences between loop iterations is computed quantitatively. Preliminary experimental results show compiler-directed thread-level speculation based on the information gathered by this technique can achieve significant performance improvement on SpMT.

1 INTRODUCTION

Speculative multithreading (SpMT) architecture has been proposed by researchers to dynamically exploit parallelism in an application [4, 7, 9, 13, 14]. In the SpMT architecture, threads can be extracted from a sequential program and speculatively executed in parallel. When the execution of parallel threads violates a data dependence specified by the sequential code, a recovery mechanism will be activated to ensure the correct sequential semantics. Furthermore, SpMT can utilize parallelism among noncontiguous regions of a program. As a result, more parallelism can be exploited than the parallelism that the compiler can usually identify statically.

Although the SpMT architecture can automatically exploit thread-level parallelism and handle recovery if mis-speculation happens, compilers play an important role in achieving maximal performance. The reason is because every recovery action incurs hefty penalty and the performance improvement gained by speculative parallelization might be nullified by the recover overheads. Compilers can avoid such performance degradation by analyzing the possibilities of conflicts between speculative threads and turning off speculation if the possibilities are over certain thresholds. Therefore, it is necessary for SpMT to incorporate a compiler that can compute quantitatively the possibilities of data and control dependences among speculation candidates in a program before execution. The goal of this work is to develop the essential analysis techniques for the SpMT compiler to compute the possibilities of dependences between speculation candidates.

Dependences between speculation candidates, such as different loop iterations or non-overlapped code regions, can be computed by comparing the read and write references between them. However, if the program contains pointer references, the possibilities of conflicts can not be computed since conventional pointer analysis techniques do not provide quantitative descriptions to tell how likely the pointers are aliased. These techniques only classify aliases or points-to relationships into must aliases or definitely-points-to relationships, which hold for all executions, and may aliases or possibly-points-to relationships, which might hold for some executions. Neither may aliases nor possibly-points-to relationships can tell how likely the conditions will hold for the executions, and consequently the compiler has to make a conservative guess and assume the conditions hold for all executions. This paper addresses this issue by presenting a probabilistic points-to analysis approach to give a quantitative description for each points-to relationship to represent the probability that it holds.

Once the probability of every points-to relationship is computed, the quantitative computation of dependences between speculation candidates can be proceeded. The results will be used to guide the thread speculation in order to reduce the impact of recovery penalties. Preliminary experimental results show compiler-directed thread-level speculation based on the information gathered by this technique can achieve significant performance improvement on SpMT.

The rest of this paper is organized as follows. Section 2 provide a description on SpMT and its cost model. Section 3 describes the probabilistic data flow analysis framework for the probabilistic points-to analysis. Section 4 details how to compute data dependence probability and the issues on applying of this information on SpMT. Experimental results...
will be presented in Section 5. Section 6 summarizes this paper.

2 SPECULATIVE MULTITHREADING

In a speculative multithreading (SpMT) model [4, 7, 9, 13, 14] threads are extracted from sequential codes and are speculatively run in parallel without violating the sequential program semantics. If there is a violation of dependence, the hardware must ensure that illegal status be recovered and mis-speculated thread re-executes with proper data. A compiler-guided speculation on multi-threading executions can reduce the probabilities of mis-speculations, and thus result in performance improvement.

2.1 Architecture and Simulator

We use an execution driven simulator SIMCA to perform experiments. SIMCA is developed by ARCTiC Lab at University of Minnesota. It is based on the SimpleScalar simulator. SIMCA simulates the hardware component interaction in the superthreaded architecture [14, 15]. The superthreaded architecture combines compiler-directed thread-level speculation of control-dependence with runtime verification of data dependence. The execution of a thread in the super-thread model is partitioned into several stages. Figure 1 shows the stages of a thread pipelining model.

In this environment, thread and data speculations can be accomplished by the abort_future instruction. When dependence is detected at runtime, it will discard the thread and allow recovery mechanisms.

2.2 Cost Model

This paper mainly focuses on the case of distributing loop iterations into threads. In this case, a compiler can use the following cost model to decide if a thread should be speculated or not.

\[ L_o > L_s + \sum_{s \in \text{set}} (V_{freq}(\text{set}) \times (O_r^{set} + L_c^{set})) \]

\[ = L_s + \sum_{s \in \text{set}} (P_{dep}(\text{set}) \times E \times (O_r^{set} + L_c^{set})) \]

\[ \simeq L_s + E \times (O_r + L_c) \times \sum_{s \in \text{set}} P_{dep}(\text{set}) \]

where set represents each violation relationship, \( L_o \) is the execution time of original codes, \( L_s \) is the time of speculative thread execution without recovery codes, \( V_{freq}(\text{set}) \) is the violation frequency for set, \( O_r^{set} \) is the overhead of set to invoke recovery codes, and \( L_c^{set} \) is the time needed to actually execute recovery codes of set. Assume \( E \) is the violation ratio when the dependence exists between threads. We have \( V_{freq}(\text{set}) \) as the multiplication of \( E \) and the probability of data dependence between threads for set, \( P_{dep}(\text{set}) \).

3 PROBABILISTIC POINTS-TO ANALYSIS

3.1 Problem Specifications

The goal of probabilistic points-to analysis is to compute the probability of each points-to relationship that might hold at every program point. For each points-to relationship, say that \( p \) points to \( v \), denoted as a tuple \( \langle p, v \rangle \), it computes the probability that pointer \( p \) points to \( v \) at every program point \( s \) during the program execution. In other words, a probability function \( P(s, \langle p, v \rangle) \) is computed for each points-to relationship \( \langle p, v \rangle \) at every program point \( s \) by the following equation

\[ P(s, \langle p, v \rangle) \text{ is the number of times } s \text{ is expected to be visited during program execution and } E(s, \langle p, v \rangle) \text{ denotes the number of times the points-to relationship } \langle p, v \rangle \text{ holds at } s \]

The probability function can be overloaded to compute the possibilities for the set of points-to relationships at every program point, if the set is represented by a vector. Specifically, if \( A \) is the set of points-to relationships at \( s \), the probability function for \( A \) at \( s \) will be

\[ P(s, A) \stackrel{\text{def}}{=} \{ P(s, \langle p, v \rangle) \mid \langle p, v \rangle \in A \} \]

\[ = \{ E(s, \langle p, v \rangle) \mid \langle p, v \rangle \in A \} \]

Such an overloaded probability function returns a vector, \( i \)th element of which contains the result of the probability function for the \( i \)th points-to relationship in \( A \).

3.2 Algorithm Outline

The conventional points-to analysis can be formulated as a data flow framework [2, 5, 8]. The data flow framework includes transfer functions, which formulate the effect of statements on points-to relationships. Suppose the sets of points-to relationships at the program points right before and after \( S \), i.e., \( S_{in} \) and \( S_{out} \), are \( IN_S \) and \( OUT_S \), respectively. Then the effect of \( S \) on points-to relationships can be represented by the transfer function \( F_S \):

\[ OUT_S = F_S(IN_S) \]

The probabilistic points-to analysis can be formulated as a data flow framework as well. If the sets \( IN_S \) and \( OUT_S \) are represented by vectors, the vector of probability functions of the points-to relationships in \( OUT_S \) can be computed by an overloaded transfer function \( F_S \):

\[ P(S_{out}, OUT_S) = F_S(P(S_{in}, IN_S)) \]

\[ = F_S(\{ P(S_{in}, \langle p, v \rangle) \mid \langle p, v \rangle \in IN_S \}) \]
$F_S$ returns a vector with the $i$th element representing the probability function of the $i$th points-to relationship in \(OUT_S\).

We formulate the process of computing the set of points-to relationships \(OUT_S\) for the basic pointer assignment statements, conditional construct, and loop construct, respectively. The probabilistic points-to-relationships will be revealed based on the data flow framework. The detailed description can be found in our research work \([6, 1]\).

## 4 DATA DEPENDENCE PROBABILITY

This section shows how to compute the probabilities of data dependences using the probabilistic points-to-analysis (PPA) information.

Consider the memory object $p$ referenced at program point $S_1$, denoted as $p_{S_1}$, and memory object $q$ referenced at program point $S_2$, denoted as $q_{S_2}$. The probability $P_{S_1, S_2}$ that $S_2$ depends on $S_1$ due to a flow dependence from $p_{S_1}$ to $q_{S_2}$ is defined as follows:

$$P_{S_1, S_2} = \frac{E(S_2, (p_{S_1}, q_{S_2}))}{E(S_2)}$$

where $E(S_2)$ is the number of times $S_2$ is executed during execution and $E(S_2, (p_{S_1}, q_{S_2}))$ is the number of times the flow dependence relationship between $p_{S_1}$ and $q_{S_2}$ holds.

A flow dependence relationship $S_1, S_2$ exists when the value of $p_{S_1}$ defined at $S_1$ flows to $S_2$ and is referenced by $q_{S_2}$. Since memory objects can be defined through a variable or a pointer, $p$ may not be defined every time $S_1$ is visited during the execution and furthermore the value of $p$ may be modified by statements between $S_1$ and $S_2$. Consequently, the information gathered by the probabilistic points-to-analysis will be used to estimate the possibility that the value of $p$ defined at $S_1$ reaches $S_2$, denoted as $P_{DEF}(p_{S_1})$. Furthermore, if $q$ is referenced at $S_2$, a true dependence relationship is generated only when $p$ and $q$ are aliases. Therefore, let $P_{REF}(q_{S_2})$ be the probability that $q$ is referenced at $S_2$ and $P_{alias}(p_{S_1}, q_{S_2})$ be the probability that $p$ and $q$ are aliases, the above equation can be computed as follows:

$$P_{S_1, S_2} = P_{DEF}(p_{S_1}) \times P_{REF}(q_{S_2}) \times P_{alias}(p_{S_1}, q_{S_2})$$

Similar formulas will be derived for anti-dependence or output dependence relationships.

## 5 EXPERIMENTS

This section first compares the estimated probabilities of all points-to-relationships by PPA with the probabilities gathered at runtime to show the accuracy of PPA. Then compiler-directed speculation will be performed on an SpMT simulator to demonstrate the impact of performance with the incorporation of dependence analysis and PPA.

### 5.1 PPA

A prototype compiler has been implemented upon the SUIF system \([3]\) and CFG library of MachSUIF \([12]\) to perform the interprocedural probabilistic points-to-analysis. Programs are first transformed from the high-SUIF format to the low-SUIF format by SUIF and then represented by CFGs using the CFG library of MachSUIF. All the variables on the CFG nodes will be associated with location sets by the SPAN routine \([11]\). The compiler will then traverse the CFGs to compute the probability function of every probabilistic points-to-relationship at each program point.

Several applications have been chosen as the benchmarks. These benchmark programs will then be executed to gather the detailed points-to-information at runtime. The runtime results will be compared with the following three variations of probabilistic points-to-analysis:

- **Probabilistic points-to-analysis based on static probabilities (PPA-S)**
  A probability will be assigned to each outgoing edge of CFG and the probabilistic points-to-analysis algorithm will be executed based on these edge probabilities. We assume $P_{IF\text{taken}} = 0.5$, $P_{IF\text{not taken}} = 0.5$, $P_{\text{LOOPback edge}} = 0.9$ and $P_{\text{LOOPexit edge}} = 0.1$.

- **Probabilistic points-to-analysis based on path profiling information (PPA-P)**
  A profiling tool has been built upon SUIF to gather the execution frequency of every edge in CFG, and probabilistic points-to-analysis will be performed based on the path profiling information to compute the probabilities of points-to-relationships in selected benchmarks.

- **Traditional points-to-analysis (TPA)**
  The probability of each points-to-relationship is assumed to be 1.

Figure 5.1 show the average errors and standard deviation of estimated probabilities of points-to-relationships by these methods compared to the profiled probabilities at runtime, respectively. The figures show the probabilistic points-to-analysis approach can estimate the likelihood that each points-to-relationship would hold with relatively small errors.

![Figure 2: The result of PPA precision test](image-url)

### 5.2 Applications on SpMT

The SIMCA simulator has been used to evaluate the performance on SpMT of several benchmark applications. First, the relation between data dependence probability and program execution times is evaluated by assuming a flow dependence exists between loop iterations with different dependence probabilities.
With the above observation, a cost model is constructed for the simulation configuration and the compiler will determine when to turn on or off the speculation mechanism. In order to evaluate the effectiveness of compiler-directed speculation, some loops are selected from programs and executed on the simulator.

The compiler will adopt the following three strategies:

**Sequential Execution** Compiler will not use speculation mechanism when it identifies a *may*-dependence. The loop body will always be executed sequentially.

**Speculation** Compiler always adopts speculation mechanism. The loop body will be executed by speculated threads.

**Probabilistic Speculation** Compiler will analyze the probabilities of dependences with the help of the PPA information. By the aid of cost model, compiler can decide whether using speculation or not.

The Figure 3 shows the comparison of execution speedup between different strategies with two thread units. The programs *en3d* and *data retrieval* have low probabilities, so it is better to use speculated threads. For programs *malloc* and *map*, they do the table lookup operations from a pointer-linked list. In most of cases, the operations are almost independent between list nodes and hence probability of conflicts is low. Consequently, the compiler turns on the speculation mechanism for these programs and achieves speedup on 2 threads.

On the other hand, the programs *shuffle* and *990127-1* exhibit high probabilities between loop iterations. Therefore the benefits from multithreading execution will be nullified by mis-speculative penalty and consequently sequential execution will be a better choice. This figure shows that the probabilistic speculative strategy uses the data dependence probability to choose the best strategy for speculation, and hence it always achieves performance improvement.

6 CONCLUSION

With the increased design of speculation mechanisms in advanced microprocessors, the abilities for compilers to be able to perform optimizations on speculations of advanced architectures become important. In this research work, we presented probabilistic point-to analysis framework and experiments which can take advantages of speculative multithreading facilities provided by architectures. The compiler with probabilistic point-to information always chooses the best strategy for speculation. Probabilistic point-to analysis will also be important for data speculations and code specializations on advanced architectures. We are in the process of investigating applications of probabilistic point-to analysis in those areas.

REFERENCES


