Software Value Prediction for Speculative Parallel Threaded Computations

Xiao-Feng Li Zhao-Hui Du Qing-Yu Zhao Tin-Fook Ngai *

Intel China Research Center, Beijing, China

* Microprocessor Research Labs, Santa Clara, California

Intel Corporation

Abstract

Despite recent advances in high performance microprocessor architecture and compilation technologies, many integer applications are still hard to speedup their performance. Value prediction and thread-level speculation are two promising techniques to discover and exploit more parallelism in applications. In this paper, we show that value prediction plays an important role in speculative parallel threaded computations. In particular, with good compilation supports, value prediction can be achieved in software without expensive hardware support. We describe this software value prediction technique and how the compiler helps to determine critical and predictable values using selective value profiling. Experiments were performed and showed that selective value profiling is effective and the software value prediction technique boosts the average performance of five speculative parallel threaded SPEC CPU2000 benchmarks by 14.3%, with the average speedup improved from 6.5% to 21.7%.

1 Introduction

Despite recent advances in high-performance microprocessor architecture and compilation technologies, many integer applications are still hard to speedup their performance. Value prediction and thread level speculation are two promising techniques to discover and exploit more parallelism in applications. These two techniques are orthogonal and can benefit from each other. Value prediction helps to break dependences that are otherwise difficult to avoid while speculative parallel threading provides more opportunities for speculation and coarser grain parallelism.

1.1 Value prediction

Value prediction has been shown to be able to provide more instruction level parallelism (ILP) by breaking data dependence [LWS96, Gab96]. The idea is straightforward: Normally, a flow of execution can proceed only when all data it needs are ready. But if the needed values can be predicted before they are really produced, then the execution can continue speculatively without stalls. The speculation results are committed when the predicted values are later proved correct; otherwise, the execution must be recovered and redirected to the correct flow that uses the correct values.

1.2 Speculative parallel threading (SPT) architecture and computation

While value prediction is helpful in advancing ILP exploitation, it is also essential for thread-level parallelism (TLP).

Along with the increasing number of transistors, current processor technologies with superscalar or VLIW can hardly bring scaled performance improvement further if limited to only fine-grained ILP. More parallelism is available at a coarser grain level, i.e. thread level. By executing piece of code ahead of time in a separate thread of control, application performance could be improved. If a thread is executed before its control dependences and data dependences are resolved, it is a speculative thread. The speculation results cannot be committed before all its assumed dependences are proved correct. This requires mechanisms to detect dependence violation and to recover from misspeculation.

1.2.1 Our speculative parallel threading execution model

In this study, we use a two-processor system to run speculative parallel threaded computations. One processor is designated as the master processor and the other is designated as the speculative processor. The master processor always executes in normal (non-speculative) mode. Each processor has its own register file and program state, while both processors share the same memory hierarchy (L1-cache can be separate but always coherent).

We have a special *fork* instruction. When the master thread executes the *fork* instruction, it forks a speculative thread that starts running at the specified address in the *fork* instruction on the speculative processor with the current thread context. The master thread continues execution in parallel with the speculative thread until it reaches the same start

address of the speculative thread. At that point, the master thread checks for any dependence violation. Depending on the results, the speculative results of the speculative thread are either committed or recovered from misspeculation. There is no register communication or synchronization between the master thread and the speculative thread during the parallel execution period.

There are two commit and recovery modes in our SPT execution model. The first mode is the *replay* mode where the master thread replays the speculated instruction one by one. The correct speculatively executed results are committed directly. Misspeculated instructions are re-executed and committed in the normal way. The second mode is the *fast-commit* mode where the master thread checks if there is any dependence violation when it reaches the speculation start point. If there is no dependence violation, all changes made by the speculative thread are committed at once; otherwise, it falls back to the replay mode.

1.3 Motivation for software value prediction

The dependences between master thread and speculative thread are critical for speculation. It is desirable if we can predict the dependences with reasonable accuracy. Value prediction appears an important means to achieve this. Since we are compiling and generating speculative parallel threaded code, we are particularly interested in value prediction techniques that do not require expensive hardware supports. We propose a software value prediction technique that can be efficient and targeted well for speculative parallel threaded compilations.

Our software value prediction is supported by our two-pass compilation process and uses selective value profiling. Different from other proposed prediction techniques [NGS99, WF97, SS98, TF01, LA00, FJL+98, FJL+98II], our technique lets the compiler to analyze the predictability and implements specific value predictor at the right place where the predicted values will be used. The value predicted for a speculative thread is pre-computed by the master thread before the speculative thread is forked. The value is communicated to the speculative thread automatically via either registers or memory.

In this software value prediction work, we use a cost model to select which variables to predict, and to guarantee the benefits of the value prediction. Selective value profiling plays a key role in the process, it helps to reduce the profiling overhead, and at the same time finds out the pattern used in the value prediction.

1.4 Paper organization

Next section we will discuss related work. In Section 3, we will give a brief introduction about our *SPT* (*Speculative Parallel Threading*) compiler framework, including the two-pass compilation process and the cost-driven SPT code transformation. We will then describe our software value prediction mechanism in the following two sections: Section 4 describes selective value profiling and Section 5 describes software value prediction and the associated implementation details. Then we present experiment results in section 6. Section 7 concludes the paper.

2 Related work

Lipasti, Wilkerson and Shen showed that load-value prediction was a promising technique to exceed data-flow limit in exploiting instruction level parallelism [LWS96]. Other work showed that the value prediction mechanism could be applied to not only load instructions, but also nearly all value-generated instructions [Gab96].

Most of current value prediction mechanisms studied in literature [NGS99, WF97, SS98, TF01] used special hardware to decide prediction and to predict values. There are four basic kinds of hardware value predictors: *Last-value* [LWS96], *Constant-stride* [GG98], *Context-based* [SS97], and *Hybrid* [WF97] predictors. The difficulty in hardware value prediction is how to find out whether an instruction is appropriate for prediction at runtime in acceptable time while keeping the hardware cost reasonable. In order to alleviate part of the problem, some work required compiler and/or profiling support such that only values that had high prediction confidence were predicted [SS98, GM97].

Fu *et al.* [FJL+98II] proposed a software-only value prediction approach that does not require any prediction hardware, and can run on existing microprocessors. It utilized branch instructions to support speculation: When the verification code found a wrong prediction, the control flow jumped to the recovery code. The prediction codes were statically inserted into the executables and were executed regardless whether the prediction was correct or if it was needed. Without special speculation hardware support, the execution context of the original code could be polluted by the prediction code. This approach showed limited performance improvement.

Fu *et al.* [FJL+98] also proposed to add explicit value prediction instructions in the instruction set architecture (ISA) for their value speculation scheduling approach.

Compiler-controlled value prediction optimization proposed by Larson and Austin [LA00] achieved better performance than previous pure software approach. It employed branch predictor for confidence estimation and used the branch prediction accuracy to resolve the value prediction accuracy problem.

Zhai *et al.* [ZCS+02] proposed compiler optimization for scalar value communication in speculative parallel threaded computation. Their compiler inserted synchronizations in critical forwarding paths in order to avoid speculation failure. Different from their approach, we do not insert synchronization to communicate correct values between threads. Instead, we try to pre-compute and predict the values for speculative thread, so that there is no extra synchronization overhead.

3 SPT compiler framework

Our work is based on the SPT compiler framework we developed on top of Intel's Open Research Compiler (ORC) [ORC]. The SPT compiler compiles a sequential program, selects and transforms appropriate loops into SPT loops, and generates the final speculative parallel threaded code. Two key features in this SPT compiler framework are its two-pass compilation process and cost-driven SPT loop selection and optimization. The software value prediction technique described in this paper takes advantage of this framework and is applied directly in the final SPT transformation. This section gives a general overview of the SPT compiler framework.

3.1 Two-pass compilation

Our SPT compilation is performed in two passes. The first pass considers every loop at each loop-nested level as an SPT loop candidate, and for each loop candidate it attempts aggressive code reordering to determine if the loop is appropriate for speculative parallel threaded execution. The results are output to a decision file that is then read back by the compiler in the second pass compilation. Based on the decision of the first pass compilation, the second pass identifies the selected loops, performs the actual SPT loop code reordering and transformation, and then generates the final code.

3.2 Cost-driven loop transformation

The SPT compiler uses a cost-driven model to guide the SPT loop transformation. For each candidate loop, it computes misspeculation cost of the loop and performs code reordering within the loop body until it achieves the minimal misspeculation cost with acceptable fork preparation work (i.e., the maximum sequential work needed before forking a speculative thread).

A loop is made a SPT loop by inserting a fork instruction in the loop body. The placement of the fork instruction effectively partitions the loop body into two regions: a *pre-fork region* before the fork instruction and a *post-fork region* after it.

Figure 1 shows a scenario that a master thread executing iteration *i* forks a speculative thread executing iteration *i+1*. It also shows the corresponding pre-fork and post-fork regions, and the possible cross-iteration dependences between the regions.

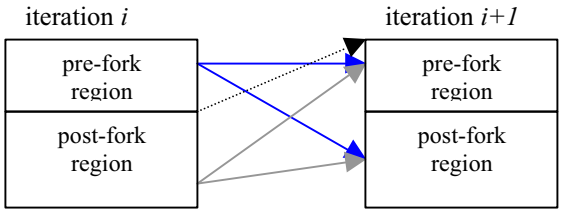

Figure 1: Scenario of SPT execution

Because the computation in the pre-fork region in iteration *i* is executed before fork, all its results are available to the speculative thread. Any cross-iteration dependence that originates from these results is always satisfied and never causes misspeculation in the speculative thread. This is why we also call the pre-fork region the *fork-preparation region*.

On the other hand, the computation in the post-fork region in iteration *i* is executed in parallel with the speculative thread. Because there is neither register communication nor synchronization between the master thread and the speculative thread, the speculative thread may read and use obsolete values existing before the forking. Any cross-iteration dependence originated from the post-fork region can be violated and causes misspeculation in the speculative thread. All computation depending on these cross-iteration dependences then needs to be re-executed later in order to recover from the misspeculation.

During SPT compilation, the compiler constructs a dependence graph of the loop body. Each node in the graph represents a separate statement of the loop body. For each data dependence or control dependence between two statements, an edge is added between the corresponding nodes. This graph is called *SPT-graph*.

Any statement in the post-fork region that is the source of a cross-iteration dependence is a *violation candidate*. The *conditional violation cost* (*CVC*) of a violation candidate is the estimated number of

instructions in the speculative thread that should be re-executed in order to recover from misspeculation, assuming the cross-iteration dependence is violated. There is a *dependence probability* (*DP*) associated with each cross-iteration dependence. It gives the probability that the dependence really occurs between two successive iterations. This dependence probability can be derived from control flow probabilities and alias probabilities, but we will not discuss the details here because of the space limit. The *misspeculation penalty* (*MSP*) of a violation candidate is therefore DP times CVC, assuming DP and CVC are independent. *Misspeculation cost* (*MSC*) of a given partition is the sum of the MSPs of all violation candidates in the post-fork region without counting a statement twice in the same re-execution instance.

The cost-driven SPT transformation partitions the loop in such a way that the misspeculation cost of the resulted partition is minimal, subjected to the constraint that the size of pre-fork region is acceptably small with respect to the loop body size. (The pre-fork region can be viewed as the sequential component of the parallel execution that limits the parallelism.) Optimal partitioning is achieved by moving the violation candidates that have high misspeculation penalty from post-fork region into the pre-fork region. As long as the pre-fork region size is acceptable, such code reordering reduces the misspeculation cost and improves the application performance.

4 Selective value profiling

As we said in Section 3, *SPT compiler framework*, the SPT loop transformation tries to find a loop partition that has minimal misspeculation cost and acceptable pre-fork region size. But in many cases, the resulted partition cannot satisfy all constraints at the same time: Either it has too big misspeculation cost or the size of its pre-fork region is too large.

For example, we find a violation candidate has unacceptably high misspeculation penalty. In order to move the statement into the pre-fork region, all statements that it depends on (which is called *depending set*) need to be moved too. This may make the size of the pre-fork region too large to comply with the partition criterion.

We find *software value prediction* (*SVP*) an effective approach to alleviate the above problem. Actually, we believe that software value prediction is essential for speculative parallel threaded computations. The basic idea is: By generating additional code in pre-fork region that predicts a key variable's value for the speculative thread, the critical dependence introduced by the variable's definition is effectively replaced by the dependence on the predicted value, which might result in more parallelism if the prediction is highly accurate. We will describe our approach in details in this and the following sections.

4.1 Critical Dependences

In order to minimize the misspeculation cost, we need to find out the *critical dependences*. In our SPT-graph, a critical dependence refers to a cross-iteration data dependence which has both unacceptably high misspeculation penalty and a too big depending set to be put into the pre-fork region.

To find out the critical dependences, according to our definition, we need the following steps:

1. Collect those dependences whose depending set size is larger than a threshold;
2. Estimate the conditional violation cost of each dependence and its dependence probability; then compute the misspeculation penalty MSP of the dependence;
3. Ignore those dependences whose MSP is less than a threshold;
4. Sort the remaining dependences in the descending order of their MSP values.

At the end, we get all the critical dependences sorted according to their criticalities.

4.2 Value-predicted dependence

The identification of critical dependences helps to determine the most beneficial candidates for value prediction. We want to replace the original critical dependences with the ones based on the predicted value, which are called *value-predicted dependences*.

A value-predicted dependence will be violated when the predicted value is mispredicted, which incurs *misprediction penalty* (*MPP*). The penalty depends on how often the prediction is wrong, and how much the misprediction costs. The former one is measured by *misprediction probability* (*MP*), which is the ratio of the number of incorrect predictions to the total number of predictions made. Given the value *prediction accuracy* (*PA*), we have MP = 1-PA. *Conditional misprediction cost* (*CMC*) is the cost incurred by a wrong prediction. CMC is basically the same as the original conditional violation cost of the replaced critical dependence. Like the misspeculation penalty, misprediction penalty is the product of conditional misprediction cost and misprediction probability, i.e., MPP = CMC * MP, assuming CMC and MP are independent.

4.3 Value profiling and pattern matching

After we have identified the critical dependences for a loop, we need to determine which variable definitions are highly predictable therefore we can apply value prediction to break the dependences. Only when the misprediction penalty is smaller than the corresponding misspeculation penalty, is the value prediction useful. If CMC equals to CVC, it means the misprediction probability should be clearly lower than the dependence probability in order to justify a dependence replacement. In other words, we must find out if the key variables exhibit some behavior that is highly software-predictable, i.e., in form of specific patterns that can be expressed easily and cost-effectively in software.

We use value profiling and pattern matching to find predictable variables. We instrument the code, run the instrumented executable to gather the generated values of the key variables being profiled, and then analyze the data by pattern matching.

Our value-pattern matching is done by a software pattern capturer called *oracle predictor*. Fed with the data from value profiling, the oracle predictor tells if a value sequence matches a prediction pattern. This oracle predictor can be run either on-line during value profiling or off-line afterwards.

The oracle predictor is flexible and extensible with many built-in patterns, such as last value, constant stride, bit shift, etc. Because it is implemented in software, not only simple but also complicated prediction patterns can be built in the oracle predictor. It tries to match the input value sequence with all pre-defined patterns in parallel, and at the end, outputs the pattern that has the highest matching ratio and the corresponding matching ratio which is later used as its prediction accuracy when the variable is selected for value prediction.

5 Software value prediction

After discovering prediction patterns by selective value profiling, we have to select which variables to be value-predicted and to generate software value prediction code for the SPT loops.

5.1 Selection of SVP variables

We determine if a variable should be value-predicted based on the relative gain from value prediction. We calculate the misprediction penalty and compare it with the original misspeculation penalty. We also compare the size of the predictor code inserted in pre-fork region with the depending set size of the original critical dependence. Only when the misprediction penalty is considerably smaller than the original misspeculation penalty and the size of the software predictor is both acceptable and apparently smaller than the original depending set size, will the variable be selected to be value-predicted.

5.2 Code transformation with SVP

Once we identify the variables to be predicted, we transform the loop into SPT form and add value-prediction code in it. Below we give a generic description of the transformation steps.

Assume we want to predict the value of variable `x` with the pattern `f(x)`. We first introduce a new variable `pred_x` to store the predicted value.

For loop iteration speculation, we compute `pred_x` at the pre-fork region of the current iteration and use it as the predicted value of variable `x` at the beginning of next iteration. To verify the value prediction, we compare the value of `pred_x` with actual value of variable `x` at the end of current iteration. If they are not equal, meaning that the value is mispredicted, we correct `pred_x` with the actual value of `x`. This guarantees the correctness of final execution of the next iteration. Figure 2 illustrates how software value prediction code is added to a SPT loop.

```
      pred_x = x;   //initialize prediction
Loop: x = pred_x;   //use prediction
      pred_x = f(x); //generate prediction
      fork(Loop);   //boundary of partition

      … = foo(x);
      x = …;

      if (x != pred_x) //verify prediction
        pred_x = x; //recover misprediction

      if (cont)  goto Loop;
```

Figure 2. Software value prediction

Note that the code introduced by software value prediction changes both the pre-fork region size and the loop body size. These two sizes are important factors in our SPT cost model. So far in our studies, these changes are found small and have negligible impact in the final SPT performance.

5.3 Safe dependence slice predictor

The above mechanism of selective value profiling and software value prediction can be extended to cover variables whose values are not directly predictable on its own but can be predicted from other known or predicted values. The key observation here is: If a variable depends on some variables that are predictable or known to be correct, the variable itself

can also be predictable using the corresponding computation relations.

The general algorithm goes this way: We first identify the variable's definition statement, which is the source node of a critical cross-iteration dependence edge. Next we analyze the SPT-graph to find out all intra-iteration dependence edges that points to this definition. If we find that all control and data dependent variables of this definition are either predictable or defined in the pre-fork region, then the variable is predictable too. We call this variable indirectly predictable. This process can be repeated recursively to find out more indirectly predictable variables that depend on indirectly predictable variables including the variable itself. To software-predict an indirectly predictable variable, we extract the dependence slice and implement a safe (i.e., non-faulting) version of the slice as the predictor.

6 Experiment results

We applied the software value prediction in our SPT compiler framework and evaluated its effectiveness. In this section we present some of the experiment results. The result data shows that selective value profiling is effective and software value prediction significantly improves performance of speculative parallel threaded computations.

6.1 Experiments and simulator

We performed experiments to evaluate the two major techniques in this work, namely, selective value profiling and software value prediction.

With the experiments that evaluate selective value profiling, we want to measure the effectiveness of our selection strategy. We also want to understand how often the oracle predictor can successfully capture a simple prediction pattern with high enough matching ratio that allows subsequent software value prediction. With the experiments that evaluate software value prediction, we want to find out how software value prediction can affect the SPT loop performance.

We used an in-house simulator for the experiments. The simulator is a data-flow IPF simulator developed for our SPT compiler research. It basically simulates an IPF pipeline and executes the SPT code sequentially. It maintains two separate clocks, separate register states and speculative execution states to keep track of the simulated parallel speculative execution. It also simulates the shared memory/cache system and performs the necessary register and memory dependence checking. This simulator is an approximate but fast simulator. Its SPT performance results have been cross-validated with another cycle-accurate but slow IPF SPT simulator. The simulator supports many different SPT configurations. Table 1 summarizes the SPT configuration used in this study.

Table 1: SPT simulation configuration

Pipeline core	Itanium2-like
Fetch/issue/writeback bandwidth	6 instructions/cycle
Commit/recovery mode	fast-commit
Thread start/fast-commit latency	6 cycles
Replay bandwidth (correctly speculated results only)	12 instructions/cycle
Cache size (byte) and latency (cycle)	L1I:16K,1; L1D:16K,1; L2:128K,5; L3:3M,12
Memory latency	150 cycles

6.2 Selective value profiling

We experimented selective value profiling with selected loops from three applications: compress95 from SPEC CPU95 and vpr, twolf from SPEC CPU2000. We chose them because the selected loops exhibit considerably different behaviors with respect to the prediction patterns.

Table 2 shows the results of the selected loops. For each application, we have three columns: *variable*, *pattern* and *ratio*. The column *variable* lists the selected variables being profiled, together with its source information such as the function name, its corresponding loop id and the source file name. The column *pattern* lists the matched pattern. It is N/A when there is no obvious matched pattern. The column *ratio* gives the corresponding matching ratio.

The data show the importance of selective value profiling: Most selected variables are critical for SPT performance and are found to be predictable with some patterns. Also note that a low matching ratio, like 0.30 for the variable out_count in compress95, does not necessarily mean that we should not apply software value prediction on it. It can be the case that its dependence probability is higher than the misprediction probability, say, 1.0 vs. 0.7 (i.e, 1 – 0.3).

Without our variable selection strategy, the compiler may need to profile far more variables than it does currently. This would incur significant profiling overhead and pattern matching difficulties, and may even make value profiling virtually impractical. More importantly, our variable selection strategy is neither tricky nor arbitrary, it actually exposes the application inherent properties that are reflected by data

Table 2: Selective value profiling experiments results

	compress95			vpr			twolf		
Input data set	input.ref			input.lite			input.lite		
Variables* selected for profiling, pattern matched and matching ratio. (* *For each variable, we also list its function name, loop id in the function and the filename*)	variable	pattern	ratio	variable	pattern	ratio	variable	pattern	ratio
	in_count, compress(), 10001, compress95.c	const-stride	1.00	**bb_index**, try_swap(), 20002, place.c	const-stride	1.00	**netptr**, countf(), 30003, countf.c	safe depend. slice	1.0
	out_count, compress(), 10001, compress95.c	const-stride	0.30	**bb_index** , try_swap(), 30003, place.c	const-stride	1.00	**termptr**, new_dbox(), 20002, dimbox.c	Safe depend. slice	1.0
	ent, compress(), 10001, compress95.c	N/A	N/A	**affected_index**, find_affected_nets(), 10001, place.c	const-stride	0.77	**termptr**, new_dbox_a, 50005, dimbox.c	safe depend. slice	1.0
	stackp, decompress(), 10001, compress95.c	last-value	1.00	**affected_index**, find_affected_nets(), 30003, place.c	const-stride	0.76	**termptr**, term_newpos(), 10001, dimbox.c	safe depend. slice	1.0
	last_ent, decompress(), 10001, compress95.c	const-stride	0.96				**termptr**, term_newpos_a(), 20002, dimbox.c	safe depend. slice	1.0
	oldcode, decompress(), 10001, compress95.c	N/A	N/A				**termptr**, term_newpos_b(), 20002, dimbox.c	safe depend. slice	1.0
							termptr , dbox_pos_2(), 20002, dimbox.c	safe depend. slice	1.0

dependence, captured by our SPT-graph, and well supported by our compilation framework.

6.3 SPT loop performance with SVP

For experiments on software value prediction, we mainly measure the SPT loop performance with and without software value prediction.

We applied software value prediction to five applications from SPEC CPU2000. Table 3 shows the performance results. All performance numbers reported in the table are the speedups of program execution times of the SPT executables on a two-core SPT machine against program execution times of the corresponding non-SPT executables on a single-core machine. All non-SPT and SPT codes were compiled at O3 optimization level with profile feedback. The row "*performance improvements without SVP*" gives the speedups achieved without application of the software value prediction technique in our SPT compilation. The row "*performance improvements with SVP*" gives the speedups with software value prediction. The last row "*contribution of SVP in total improvements*" shows the contribution proportions of the software value prediction technique in the total improvement achieved by our SPT compilation.

From the experiment results, we can clearly see the significance of software value prediction in SPT transformation. For some applications, e.g. mcf and twolf, the contribution of SVP in the total improvement is higher than 90%. Overall, the contributions are more than 50%. This also justifies our philosophy of two-pass compilation and selective value profiling. Software value prediction does not necessarily require selective value profiling. However, all results we show here are obtained with selective value profiling. We are still exploiting other software value prediction schemes. Nevertheless, we believe value profiling is an essential means to discover predictable variables.

Table 3: Effects of software value prediction

Application	gzip	vpr	mcf	parser	twolf
perf. imprv. without SVP(%)	13.7	11.1	1.0	6.4	0.4
perf. imprv. with SVP(%)	17.1	29.0	15.1	22.8	24.4
contrib. of SVP in total imprv.(%)	19.9	61.7	93.4	71.9	98.4

7 Conclusion

Value prediction and thread-level speculation are two promising techniques to discover and exploit more parallelism in applications. In this paper, we described two major techniques, namely selective value profiling and software value prediction, for speculative parallel threaded computations. By taking advantage of our two-pass SPT compiler framework, critical dependences that potentially lead to big misspeculation penalty are identified and selectively value-profiled to determine if the associated variables can be value-predicted. Then highly predictable variables are fed back to the compiler and the compiler implements the value prediction in software in the final speculative parallel threaded code.

Experiments were performed to evaluate the effectiveness of selective value profiling and the performance of software value prediction. The results showed that value prediction can be done without expensive value prediction hardware supports, and the proposed selective value profiling and software value prediction techniques are effective and efficient. Five SPEC CPU2000 benchmarks were compiled to generate speculative parallel threaded code with and without software value prediction. The performance evaluation results showed that the software value prediction boosts the average performance of five speculative parallel threaded benchmarks by 14.3%, with the average speedup improved from 6.5% to 21.7%.

As shown in this study, software value prediction is an important and promising technique. We are currently studying how to extend this basic idea to cover more value predictable cases, perfecting our SPT compilation and performing more extensive evaluations. One important future work is to study the influence of the software value prediction technique on the SPT algorithms used in our compilation framework. This includes the cost of value profiling and the construction of the oracle predictor.

References

[CRT99] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction", International Symposium on Computer Architecture, 1999.

[FJL+98] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value Speculation Scheduling for High Performance Processors", in 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[FJL+98II] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Software-Only Value Speculation Scheduling", Technical Report, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC27695-7911, June 1998

[Gab96] F. Gabbay, "Speculative execution based on value prediction", Technical Report 1080, Department of Electrical Engineering, Technion-Israel Institute of Technology, 1996.

[GG98] J. Gonzalez and A. Gonzalez; "The Potential of Data Value Speculation to Boost ILP"; International Conference on Supercomputing, 1998

[GM97] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction", International Symposium on Microarchitecture, 1997.

[LWS96] M. Lipasti, C. Wilkerson and J. Shen, "Value Locality and Load Value Prediction", International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.

[LA00] E. Larson and T. Austin, "Compiler Controlled Value Prediction Using Branch Predictor Based Confidence", ACM/IEEE 33rd International Symposium on Microarchitecture (MICRO-33), December 2000.

[MG00] P. Marcuello and A. Gonzalez, "A Quantitative Assessment of Thread-Level Speculation Techniques", Proc. of the 1st. Int. Parallel and Distributed Processing Symposium (IPDPS'00), Canc? (Mexico), May 1-4, 2000.

[MTG99] P. Marcuello, J. Tubella, and A. Gonzalez; "Value Prediction for Speculative Multithreaded Processors"; International Symposium on Microarchitecture, 1999.

[NGS99] T. Nakra, R. Gupta, and M.L. Soffa; "Global Context-Based Value Prediction"; International Symposium on High-Performance Computer Architecture, 1999.

[ORC] Open Research Compiler, Intel Co. Ltd., http://ipf-orc.sourceforge.net/.

[SS97] Y. Sazeides and J. Smith; "The Predicatibility of Data Values"; International Symposium on Microarchitecture, 1997.

[SS98] Y. Sazeides and J. Smith, "Modeling Program Predictability", International Symposium on Computer Architecture, 1998.

[TF01] R. Thomas and M. Franklin, "Using Dataflow Based Context for Accurate Value Prediction", Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2001.

[WF97] K. Wang and M. Franklin; "Highly Accurate Data Value Prediction using Hybrid Predictors"; International Symposium on Microarchitecture, 1997.

[ZCS+02] A. Zhai, C. B. Colohan, J. G. Steffan and T. C. Mowry, “Compiler Optimization of Scalar Value Communication Between Speculative Threads”, The Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, CA, USA, Oct 7-9, 2002.