Exploiting Thread-Level Speculative Parallelism with Software Value Prediction

Abstract

Software value prediction (SVP) is an effective and powerful compilation technique helping to expose thread-level speculative parallelism. With the help of value analysis and profiling, the compiler identifies critical and predictable variable values and generates speculatively parallelized programs with appropriate value prediction and misprediction recovery code. In this paper, we examine this technique in detail, describe a complete and versatile SVP framework and its detailed implementation in a thread-level speculative parallelization compiler, and present our evaluation results with Spec2000Int benchmarks. Our results not only confirm quantitatively that value prediction is essential to thread-level speculative parallelism, they also show that the corresponding value prediction can be achieved efficiently and effectively by software. We also present evaluation results of the overhead associated with software value prediction and the importance of different value predictors in speculative parallel loops in Spec2000Int benchmarks.

1. Introduction

Recent studies show value prediction is a promising technology to break the data-flow parallelism limit [5],[11]. While most of the literatures on value prediction focused on *instruction-level parallelism* (ILP), our research in value prediction compilation technology shows that, value prediction is actually essential in exploring *thread-level parallelism* (TLP), and can be done purely in software with speculative multithreaded processors.

1.1 Thread-level Speculation (TLS)

To exploit good TLP with existing programming languages, a promising approach is to use thread-level speculation (TLS), with which a piece of code (the speculative thread) runs ahead of time speculatively in parallel with the code it depends on without committing its computation results until its assumed input data are proved correct. Correctness of input data decides the delivered TLP performance.

Value prediction can be applied here to improve the performance by predicting the input data for the speculative thread. We developed a novel technique to achieve such value prediction efficiently and effectively in software. Our *software value prediction* (SVP) technique is implemented and evaluated in a speculative parallel threading (SPT) framework [2]. Before we describe the complete SVP methodology and its implementation in details, we first give a brief introduction of the SPT execution model and its application to loops.

1.1.1 The SPT Execution Model

In the SPT execution model that we studied [9], there are two processing units. One is designated to be the main processor that always runs in normal (non-speculative) mode. The other is speculative processor that runs in speculative mode. When the main thread on the main processor executes a special instruction *spt_fork*, a speculative thread is spawned with the context of the main thread on the speculative processor and starts running at the instruction address specified by *spt_fork*. All execution results of the speculative thread are buffered and not part of the program state. After executing the *spt_fork* instruction, the main thread continues its execution in parallel with the speculative thread. There is no direct communication or explicit synchronization between the two threads except they share the memory hierarchy. The instruction address of *spt_fork* in main thread is called *fork-point*; the address specified by *spt_fork* where the speculative thread begins its execution is *start-point*.

When the main thread arrives at the *start-point*, i.e., the place where the speculative thread starts the speculative execution, it will check the execution results of the speculative thread for any dependence violation. Depending on the check result, the main thread takes either of the following actions: If there is no dependence violation (i.e., no misspeculation), the entire speculative state is committed at once. Otherwise, the main thread will re-execute the misspeculated instructions while committing the correct execution results.

1.1.2 SPT Loop and Partition

We apply the SPT execution model to exploit loop-level parallelism. That is, when an iteration of a loop runs on the main processor, a speculative thread will be spawned on the speculative processor to run the successive iteration. We call this kind of loop an *SPT loop*. The compiler's job is to generate SPT loops automatically and guarantee the SPT loops bring performance benefits. Figure 1 illustrates the execution scenario of an SPT loop.

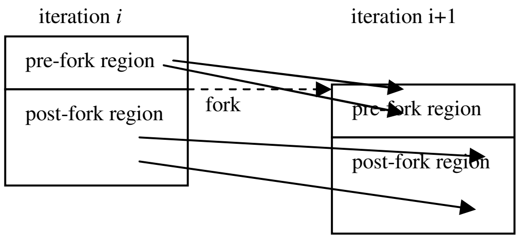


Figure 1: *SPT loop partition and execution scenario.*

(The arrows show the true dependences between the iterations. The middle line in the loop body refers to the fork-point, and the start-point is always at the beginning of the loop body.)

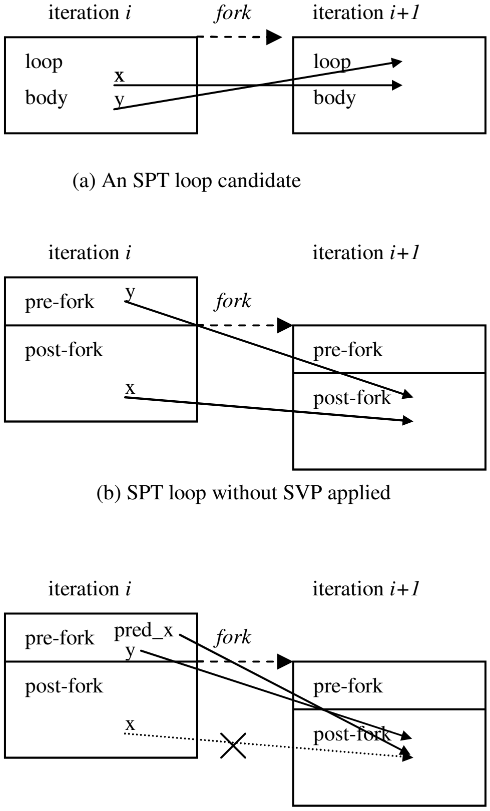
In Figure 1, the main thread executing iteration *i* forks the speculative thread for iteration *i+1*. The insertion of the instruction *spt_fork* effectively partitions the loop body into two regions: a *pre-fork* region and a *post-fork* region. The true data dependences between two consecutive iterations are shown as arrows. We call the source of any cross-iteration dependence a *violation candidate*.

Since the speculative thread is spawned after the main thread has finished its pre-fork region, all dependences originated from the pre-fork region to the speculative thread are guaranteed to be satisfied. We care only those dependences originated from the post-fork region of the main thread. In order to avoid dependence violation, the compiler can try to reorder all violation candidates into pre-fork region. However, such code reordering will be restricted by the dependences between the pre-fork and the post-fork regions within the iteration. Furthermore, the pre-fork region is part of the sequential component of the parallel execution of an

SPT loop. Amdahl's law requires the pre-fork region size small enough compared to the post-fork region in order to bring any parallelism benefits. Since the *start-point* of an SPT loop is always at the beginning of the loop body, the compiler's work for the SPT loop parallelization is essentially to perform desirable code transformations on the loop body and then determine the optimal *fork-point* within the loop body.

1.2 Software Value Prediction (SVP)

Value prediction techniques can be used to improve speculative thread-level parallelism. When a variable accessed by the speculative thread is probably to be modified by the main thread, we can predict the modified value for the speculative thread before its first access. If the value is correctly predicted, the corresponding data dependence is effectively removed, with certain overhead for the prediction though.



(a) An SPT loop candidate

(b) SPT loop without SVP applied

(c) SPT loop with SVP

Figure 2: *Illustration of SVP in SPT loop*

In our work, value prediction is conducted in pure software without any special value prediction hardware support. The idea of *software value prediction* (SVP) is to have the compiler determine which values are critical for performance and predictable, then inserting the prediction statements (predictors) in proper places in the compiled code. Figure 2 illustrates how SVP is applied in SPT loop to improve the thread-level parallelisms. Figure 2(a) is a loop selected as SPT loop candidate; it has two violation candidates that defines values of variables x and y for next iteration. In Figure 2(b), the loop is transformed into an SPT loop where the violation candidate for variable y is moved into pre-fork region. Variable x's definition cannot be reordered into pre-fork region because of either the intra-iteration dependence or the pre-fork region size constraints. Figure 2(c) shows the defined value of variable x is predicted in pre-fork region. The original dependence from x is virtually replaced by the new dependence originated from variable pred_x.

Different from the case of variable y, the value of pred_x may be mispredicted, causing misspeculation on variable x. The value misprediction will be captured by the software checking code inserted by the compiler, which we will explain next.

1.2.1 An example of software value prediction

Figure 3 gives a more concrete example of the software value prediction technique.

Consider the loop in Figure 3(a). It is very difficult to get any loop-level parallelism without any special transformation because there are cross-iteration dependences incurred by variable x. Even with speculation, if the speculative thread uses old x value to compute foo(x) and bar(x), its speculative execution is useless. In this case, the compiler can profile the value x. Assuming it finds that x is often incremented by 2 by bar(x), the compiler can predict x by inserting the predictor statement "pred_x = x+2;" before the *spt_fork* instruction as shown in Figure 3(b). The predicted value is used as the initial value of x by the speculation thread with statement "x = pred_x;". The check and recovery code "if(pred_x != x) pred_x = x;" is inserted at the end of loop.

```
                             pred_x = x
                             while(x){
                              start:
                                x = pred_x;
                                pred_x = x+2;
                                spt_fork(start);
while(x){                       foo(x);
   foo(x);                      x = bar(x);
   x = bar(x);                  if(pred_x != x)
}                                   pred_x = x;
                             }


(a) Original loop            (b) SVP-transformed loop
```

Figure 3*: Example of SVP compilation technique*

If the predicted value pred_x does not match with the new value of x, the assignment statement "pred_x = x;" will be executed. As the variable pred_x becomes modified after the fork-point, it will be caught by the SPT dependence check as a dependence violation and triggers misspeculation recovery according to the SPT execution model. The re-execution of the next iteration is guaranteed to be correct because the value of pred_x has been corrected by "pred_x = x;".

Our study showed that software value prediction can boost thread-level speculation performance significantly. We have developed a complete and versatile SVP framework and implemented it in our SPT compiler. Our evaluation results confirm quantitatively that value prediction is essential to thread-level speculative parallelism. This paper examines the SVP technique in details, describes our general SVP framework and reports its effectiveness and efficiency in improving thread-level parallelism for the SPECint2000 benchmark suite.

1.3 Paper Organization

The rest of the paper is organized as following. Section 2 discusses the related work; then we briefly introduce the SPT compilation framework for loop iteration-based speculative thread-level parallelism in Section 3. Our general SVP framework and its detailed implementation in the SPT compiler are described in Section 4 and Section 5 respectively. Section 6 evaluates the SVP technique with its performance results and overhead data. We conclude the paper in Section 7.

2. Related Work

Lipasti, Wilkerson and Shen showed that load-value prediction is a promising technique to exceed data-flow limit in exploiting instruction level parallelism [11]. Other work showed that the value prediction mechanism could be applied to not only load instructions, but also nearly all value-generated instructions [5]. Most of current value prediction mechanisms studied in literatures [15],[19],[20],[21],[8] use special hardware to decide prediction pattern and to predict values. Basically there are four kinds of hardware value predictors: Last-value [11], Constant-stride [5], Context-based [18], and Hybrid [21]. The difficulty in hardware value prediction is in finding out whether an instruction is appropriate for prediction at runtime within acceptable period while keeping the hardware cost reasonable. In order to alleviate the severity of the problem, some work [7], [19] require compiler and/or profiling support such that only values that have high confidence are predicted.

Fu et al. [4] proposed a software-only value prediction approach that does not require any prediction hardware, and can run on existing microprocessors. It utilizes branch instruction for speculation: When the verification code finds a wrong prediction, the control flow jumps to the recovery code. The prediction codes are statically inserted into the executables and are executed regardless whether the prediction is correct or necessary. 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. [3] also proposed to add explicit value prediction instructions in the instruction set architecture (ISA) for their value speculation scheduling.

Compiler-controlled value prediction optimization done by Larson and Austin [12] achieved better performance than previous pure software approach. It employs branch predictor for confidence estimation and uses the branch prediction accuracy to resolve the value prediction accuracy problem.

Zhai et al. [22] 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 there is no extra synchronization overhead while may have misprediction penalty.

Steffen et al.[23] have proposed techniques to improve value communications in TLS computation, including value prediction for both memory values and forwarded values. They found predictors must be throttled to target only those dependences that limit performance, and squashing the prevalent silent stores can greatly improve the TLS performance. In SVP, the compiler only inserts predictors for the variables that are sure to bring performance benefits with our cost model. The last-value predictors in SVP is very similar to the silent stores, and we drew the same conclusion that last-value predictor contributes a large portion to the performance improvement.

Cintra and Torrellas[24] has proposed a complete hardware framework for handling data dependence violations in speculative parallelization. The techniques include value prediction for same-word dependences. They conducted the experiments with a suite of floating-point applications, and found that values are either highly unpredictable or do not change because they are accessed by silent stores. Our work with SVP is evaluated with SPECint2000 benchmarks, which shows value prediction is effective to eliminate squashes due to data dependences.

3. A Compilation Framework for Thread-Level Speculative Parallelization

We have developed a comprehensive compiler framework based on Open Research Compiler (ORC [16]) to extract loop-based thread-level speculative parallelism by executing successive iterations in parallel threads [2]. We developed a misspeculation cost model and used it to drive the speculative parallelization. The framework allows the use of enabling techniques such as loop unrolling, variable privatization and dependence profiling to expose more speculative thread-level parallelisms. Our software value prediction technique is implemented and evaluated in the framework. In this section, we give a brief description about our SPT compilation framework.

3.1 Cost-driven Compilation

The goal of SPT compiler is to find out the optimal loop partition for each loop candidate so that the generated SPT loop has minimal misspeculation cost. SPT compiler accomplishes the goal with its cost-driven compilation framework in a two-pass compilation process, as is described in this section.

In order to estimate the misspeculation cost, we introduce the concepts of *dependence probability* and *misspeculation penalty*. Dependence probability $Probability_{Dependence}(dep)$ of a loop-carried dependence *dep* is the probability of the dependence really happening at runtime. It can be computed as the ratio of the number of iterations when the dependence happens to the number of total iterations the loop executes. The misspeculation penalty $Penalty_{Dependence}(dep)$ of the dependence is the misspeculated instruction count caused by the dependence and need to be re-executed. The *misspeculation penalty* caused by the dependence is computed as equation E1:

$$Penalty_{Misspeculation}(dep) = Penalty_{Dependence}(dep) * Probability_{Dependence}(dep) \quad \textbf{(E1)}$$

And the overall effects of all the dependences in a loop is computed as *misspeculation cost,* which is expressed conceptually[1] as equation E2,

$$Cost_{Misspeculation} = \sum_{dep} Penalty_{Misspeculation}(dep) \quad \textbf{(E2)}$$

In order to select and transform only good SPT loops without missing any good ones, SPT compiler goes through two compilation passes. The first pass selects loop candidates according to simple selection criteria like loop body size and trip count, and apply loop preprocessing such as loop unrolling and privatization for more opportunities of thread-level parallelism. Next, the SPT compiler finds out the optimal loop partition for each loop candidate, and determines its potential speculative parallelism amount. The optimal partition results of all loop candidates are output and the first pass finishes without any real

[1] It is conceptual because the different dependences may cause same set of instructions to be re-executed. Please refer to [2] for detail computation of misspeculation cost.

permanent transformation. Then the second pass reads back the partition results and evaluates all loops together, selects all good and only good SPT loops. These loops are again preprocessed, partitioned, and transformed so as to generate final SPT code.

3.2 Optimal Loop Partition Searching and SPT Loop Transformation

Since a partition is decided by the set of statements in post-fork region (or pre-fork region because they are complementary), the misspeculation cost of a given partition is decided uniquely by the violation candidates in the post-fork region. That means a combination of violation candidates can uniquely decide a loop partition. So SPT compiler searches for the optimal loop partition of an SPT loop candidate by enumerating all the combinations of violation candidates, computing each combination's misspeculation cost assuming only the violation candidates in the combination are in post-fork region, and selecting the combination that has minimal cost. The actual search space is greatly reduced with bounding functions and heuristics [2].

After the first compilation pass, we obtain the optimal partition and its associated optimal misspeculation costs for each loop candidate. In the second compilation pass, our speculative parallelization examines all loop candidates together (such as all nesting levels of a loop nest) and select all those good SPT loops that likely bring performance benefits and does the final SPT transformation to generate optimal SPT loop code.

With the resulted optimal partition of a good SPT loop candidate, SPT compiler transforms the original loop to generate the expected partition by code reordering: Assuming the SPT_FORK statement (which is compiled into the *spt_fork* instruction later) is initially inserted at the loop body start, the transformation work is to move the violation candidates in the pre-fork region above the SPT_FORK statement.

```
while( c!= NULL ){
    c1 = c->next;
    free_Tconnector(c->c);
    xfree(c, sizeof(Clause));
    c = c1;
}
(a) original loop
while( c!= NULL ){
start:
    c = temp_c;
    c1 = c->next;
    temp_c = c1;
    SPT_FORK(start);
    free_Tconnector(c->c);
    xfree(c, sizeof(Clause));
    c = c1;
}

(b) SPT-transformed loop
```

Figure 4: *Example of SPT transformation (without SVP)*

Figure 4 is a real example of SPT loop transformation from *parser* application in SPEC2000int benchmarks. In the example, the *start-point* is indicated by the label `start`. The loop body is partitioned by the statement `SPT_FORK(start)`. The violation candidate the is moved to pre-fork region is statement "`c = c1;`". Since variable `c1` depends on statement "`c1 = c->next;`", both are put in to pre-fork region. In order to maintain the live-range of variable `c`, another variable `temp_c` is introduced.

4. SVP Methodology in SPT Compiler

Software value prediction can be achieved systematically and effectively with SVP methodology we developed in SPT compiler. In this section, we use a practical example to explain the methodology. Our SVP implementation in SPT compiler will be described in Section 5, "SVP Implementation in SPT Compiler".

Figure 5 is a code snippet in application *mcf* from SPECint2000 benchmark, which is a hot loop in function `price_out_impl()`, covering 21% of total running time with *train* input set. It is easy to realize that, this loop can barely deliver any iteration-based thread-level parallelisms if without any transformations because of the cross-iteration dependence incurred by variable `arcin` and `newarcs`. We will describe how SVP can improve the achieved parallelisms.

SPT compiler firstly identifies all the violation candidates, and estimates the misspeculation cost of them. For those dependences that have significantly impact on the delivered TLP, the compiler will figure out if there are opportunities to apply SVP technique, by predicting the dependence source variables. These variables are called *critical variables*, such as variable `arcin` and `new_arcs` in Figure 5

```
while( arcin ){
   tail = arcin->tail;
   if( tail->time + arcin->org_cost > latest ){
      arcin = (arc_t *)tail->mark;
      continue;
   }
   red_cost = compute_red_cost( arc_cost, tail,
                         head_potential );
   if( red_cost < 0 ){
      if( new_arcs < MAX_NEW_ARCS ){
            insert_new_arc( arcnew, new_arcs, tail,
                    head, arc_cost, red_cost );
            new_arcs++;
      }
      else if( (cost_t)arcnew[0].flow > red_cost )
            replace_weaker_arc( arcnew, tail, head,
                       arc_cost, red_cost );
   }
   arcin = (arc_t *)tail->mark;
}
```

Figure 5: *Example loop in `mcf` from SPECint2000*

4.1 SPT Cost Model with SVP

The idea of software value prediction can fit in our SPT cost model described in Section 3.1 *Cost-driven Compilation* very well. For example in Figure 2, the original dependence caused by variable `x` in the loop without SVP is replaced by a new dependence originated from the predicted variable `pred_x` in pre-fork region. Since the prediction code in the main thread is executed before forking, the dependence from prediction can always be satisfied. The predicted value can be wrong in some iterations, which is characterized by the *probability of misprediction*; and the *misprediction penalty* of a dependence *dep* is the product of the misprediction probability and its dependence penalty shown as equation E3.

$$Penalty_{Misprediction} = Penalty_{Dependence}(dep) * Probability_{Misprediction}(dep) \quad \textbf{(E3)}$$

As long as the misprediction probability is much smaller than the replaced dependence probability and the predictor code size is small, SPT compiler can apply the prediction in loop transformation[2].

With the SVP cost model, it's clear that SVP does not try to reduce the dependence penalty of any dependence; instead, it tries to reduce the dependence probability to misprediction probability. Next we show how the model directs the software value prediction.

After the compiler identifies the critical variables of the loop, SPT compiler needs to know the proper prediction pattern of each variable and the corresponding misprediction probability.

4.2 Selective Value Profiling

Prediction pattern of the critical variables could be found by program analysis, but it does not always work. For instance, neither `arcin` nor `new_arcs`' prediction pattern can be derived by program analysis because of complicated control and data flow. SVP uses value profiling for prediction pattern identification.

Only critical variables are profiled. The compiler instruments the application with hooks into a value profiling library, which has some built-in prediction patterns, such as:

- *constant*, where the variable has only one fixed value;
- *last-value*, where the variable has the same value as last time it is accessed;
- *constant-stride*, where the variable's value has constant difference with the value last time it is accessed;
- *bit-shift*, where the variable's value can be got by shifting its last value in constant bits;
- *multiplication*, where the variable's value is constant times of its last value;

The instrumented executable runs with typical input set for value profiling. During the value-profiling run, the runtime values of the critical variables are fed into the library, where they are matched with the built-in prediction patterns, and the matching results are recorded. In the end of the execution, the results are output into a feedback file which has the SVP-needed information for each critical variable, such as the best matched prediction pattern, the matching ratio with the pattern, and total hit counts, etc. The misprediction probability of the pattern is equal to (1 – matching ratio).

As to our example, when run with *train* input set, variable `arcin` shows constant-stride pattern with constant "`-192`" and perfect matching ratio 100%, while `new_arcs` shows last-value pattern with matching ratio 99%.

SPT compiler estimates the overall effects of the penalties of dependence or prediction of all violation candidates and decides which variables are to be predicted according to the cost model. Here both `arcin` and `new_arcs` are selected to be predicted.

4.3 SVP Code Transformation

After the predictable variables are selected, the compiler inserts the predictor statement in the loop by code transformation.

SVP transformation can be accomplished basically in four steps, although the real work depends on the speculative architecture and application model. Assume the *fork-point* and *start-point* is decided already and we are going to insert a predictor for variable `x` which has prediction pattern `predictor(x)`. A new variable `pred_x` is introduced to store the predicted value for the variable `x` in the master thread. The transformation steps are:

1. **Prediction initialization**: Insert an assignment "`pred_x = x;`" before the loop, which prepares the predicted value;
2. **Prediction use**: Insert an assignment "`x = pred_x;`" in the beginning of the loop body, which consumes the predicted value;
3. **Prediction generation**: Inserts an assignment "`pred_x = predictor(x);`" after the prediction use statement, which generates the predicted value and saves it in `pred_x`;
4. **Prediction verification**: Inserts statement "`if(pred_x!=x) pred_x = x;`" in the end of the loop body, which checks the correctness of the prediction and corrects it if it is wrong.

```
pred_arcin = arcin;   //prediction intialization
while( arcin ){
 start:                //start-point
  arcin = pred_arcin; //prediction use
  pred_arcin = arcin -192; //prediction generation
  SPT_FOOK(start);        //fork-point
  tail = arcin->tail;
  if( tail->time + arcin->org_cost > latest ){
    arcin = (arc_t *)tail->mark;
    continue;
  }
  red_cost = compute_red_cost( arc_cost, tail,
                          head_potential );
  if( red_cost < 0 ){
    if( new_arcs < MAX_NEW_ARCS ){
         insert_new_arc( arcnew, new_arcs, tail,
                   head, arc_cost, red_cost );
         new_arcs++;
    }
    else if( (cost_t)arcnew[0].flow > red_cost )
         replace_weaker_arc( arcnew, tail, head,
                        arc_cost, red_cost );
  }
  arcin = (arc_t *)tail->mark;
  if( pred_arcin!=arcin )//prediction verification
       pred_arcin = arcin; //misprediction recovery
}
```

Figure 6: Example loop in `mcf` with SVP transformation

The SVP-transformed code for the example loop is shown in Figure 6. Careful readers may find that we do not predict variable `new_arcs` with last-value predictor in the transformed code. The reason is, its last value as part of the thread context is copied to the speculative thread naturally as the SPT architecture supports. If the architecture model does not support context copy, SVP needs to predict it as well. These issues are important for a practical implementation.

In our experiment with the example loop, the transformed one gets 49.76% performance speedup compared to the original one, which alone increases *mcf's* overall performance by 10% because of the big coverage of the loop.

4.4 Advanced SVP Techniques

In our experiments, the methodology described so far is enough in handling most of the cases in SPECint2000 benchmarks. In this subsection, we introduce more sophisticated techniques for software value prediction in order to exploit more parallelisms.

4.4.1 Indirect Predictor

One situation we meet is that not all critical variables exhibit obvious prediction patterns, for instance, a variable storing memory pointer to the nodes of a tree, whose runtime values are pretty randomly scattered in the heap space.

Figure 7 shows a similar but different situation, which is a loop in function `compress_block()` of SPECint2000 application *gzip*. In this loop, variable `dx` is critical for iteration-based

[2] The actual decision as to whether replace a dependence with prediction is made by computing the overall effect of the dependence or prediction penalties of all violation candidates.

speculative parallelisms, but its value does not have good predictability.

On the other hand, dx's value depends on another critical variable flag by control-dependence shown in bold fonts in the figure. If flag has very good predictability, we can predict dx's value based on flag's prediction as long as the control-dependence has reasonable probability. This kind of predictor is identified by combining program analysis technique with software value prediction, and is called *indirect predictor* for distinction from the previous *direct predictor*.

```
do {
   if ((lx & 7) == 0) flag = flag_buf[fx++];
   lc = l_buf[lx++];
   if ((flag & 1) == 0) {
      send_code(lc, ltree);
      Tracecv(isgraph(lc), (stderr," '%c' ", lc));
   } else {
      code = length_code[lc];
      send_code(code+LITERALS+1, ltree);
      extra = extra_lbits[code];
      if (extra != 0) {
          lc -= base_length[code];
          send_bits(lc, extra);
      }
      dist = d_buf[dx++];
      code = d_code(dist);
      Assert (code < D_CODES, "bad d_code");
      send_code(code, dtree);
      extra = extra_dbits[code];
      if (extra != 0) {
          dist -= base_dist[code];
          send_bits(dist, extra);
      }
   }
   flag >>= 1;
} while (lx < last_lit);
```

Figure 7: *Indirect predictor example in* gzip *application*

In some cases, the critical variable is predictable by both direct and indirect predictor. The compiler needs to choose either of them to apply the code transformation. Predictor selection is then an interesting topic that we will describe in our SVP implementation in Section 5.

4.4.2 Speculative Precomputation

Another important technique in SVP compilation is *speculative precomputation*, which predicts a critical variable with code slice extracted from the program without depending on any other variable's prediction.

Let's take an example from SPECint2000 application *twolf's* function term_newpos() shown in Figure 8 .

```
for( termptr = antrmptr ; termptr ;
      termptr = termptr->nextterm ) {

   ttermptr = termptr->termptr ;
   ttermptr->flag = 1 ;
   ttermptr->newx=termptr->txpos[newaor/2] + xcenter ;

   dimptr = netarray[ termptr->net ] ;
   if( dimptr->dflag == 0 ) {
      dimptr->dflag = 1 ;
      dimptr->new_total = dimptr->old_total +
                  ttermptr->newx - ttermptr->xpos ;
   } else {
      dimptr->new_total+=ttermptr->newx-ttermptr->xpos;
   }
}
```

Figure 8: *Speculative precomputation example in* twolf

In this loop, variable termptr is highly critical for correct speculation, but its value exhibits very low predictability. And because of the probable alias between termptr and other modified pointer variables in the loop body, the compiler cannot guarantee "termptr = termptr->nextterm;" alone can generate correct value for next iteration.

On the other hand, the compiler finds the alias probability between termptr and other pointer variables are small by value profiling or other profiling tools if available. It can extract the code "termptr = termptr->nextterm;" from termptr's all possible definition statements, use it as the predictor so as to speculatively precompute termptr's value for the speculative thread.

We find in our experiments that sometimes the predictor is the combination of speculative precomputation and the direct prediction. We regard this also a kind of indirect predictor.

4.4.3 Prediction Plan

Sometimes there are multiple critical variables that need predicting for one speculative thread, the compiler cannot simply group their predictors to be the prediction code, because these critical variables may be inter-dependent or their predictors may share some codes. A simple combination of the best predictors for each critical variable does not necessarily mean optimal prediction code, it is important to find out the combination that can achieve best overall TLP performance.

The prediction code for a speculative thread, as a reasonable combination of all the predictors, is called a *prediction plan*. It is desirable to find out the optimal prediction plan under our cost model that can bring minimal misspeculation cost with reasonable code size. We have developed such a plan-searching algorithm in our SVP implementation discussed in next section.

5. SVP Implementation in SPT Compiler

We introduced the SVP methodology in SPT compiler; now in this section we describe the implementation details of software value prediction about its critical variable identification and optimal prediction plan searching.

5.1 Critical Variable for Value Profiling

Since the predicted value is only used by the speculative thread in our model, it is irrelevant to the semantic correctness of the application. It could be possible to profile as more variables as possible and use more complicated prediction patterns in order to achieve better performance; but we choose profile only some critical variables because of the two negative impacts caused by more profiled variables.

Although the speculation model guarantees the mispredicted values not affect the correctness, they do affect the performance. Prediction code run in the pre-fork region of the master thread is executed serially hence consumes precious processor cycles; too big size of the prediction code may negate our goal of increasing thread-level parallelisms. There is a tradeoff between the prediction code size and its reduced misspeculation penalty.

Actually SPT compiler has two thresholds for loop partitioning to count into the balance of pre-fork region size and misspeculation cost, that is, the pre-fork region size can never exceed size *S* and the misspeculation cost should be smaller than threshold *C*. Both thresholds are constants during the compilation as a ratio value to the loop body code size and execution time.

The code reordering in SPT compilation cannot increase the pre-fork region size to exceed *S* by moving a violation candidate into the pre-fork region, even the resulted loop partition has higher than *C* misspeculation cost. That is why the compiler fails to move the violation candidate x into pre-fork region in Figure 2, even if it has high misspeculation penalty; and that is why value prediction is important: It can reduce the misspeculation cost of a violation candidate without moving it into pre-fork region like it does with variable `x` in Figure 2. Actually in our evaluation with SPECint2000, we believe value prediction is essential for ultimate TLP performance.

SPT compiler introduces *dependence slice* in order to accurately estimate the code size for a violation candidate, which refers to the statements or expressions dependent by the violation candidate between *fork-point* and *start-point*, i.e., the loop body in our implementation.

As discussed above, when the dependence slice is very small, it could be cloned in the pre-fork region to precompute the value, which can be implemented by the speculative precomputation technique developed in SVP, while the precomputed value is always correct. On the other hand, since SPT compiler itself can reorder the small dependence slice to pre-fork region, there is no need to precompute it at the beginning. In any case, SVP will not profile this kind of defined variables.

If a variable has big dependence slice, it still does not necessarily mean to be profiled by SVP. One reason is the misspeculation penalty caused by the violation candidate can be very small compared to the cost threshold *C*. The other situation it is not profiled is, when the variable directly depends on another defined variable, we need profile only the latter one, and derive the former variable's value from their relation. This results with indirect predictor as we described in Subsection 4.4.1. In this case, the misprediction probability of the indirect predictor is derived by probability propagation in the dependence slice from the direct predictor.

Then all remaining critical variables for value profiling have high misspeculation cost and big dependence slice size, and we hope to predict them directly.

5.2 Prediction Plan Search

With the prediction pattern chosen for each critical variable by the profiling library, the compiler will decide how to predict it by prediction plan searching. We use a branch-bound algorithm in our implementation to search for the optimal prediction plan. The pseudo-code of the algorithm is shown in function `predictor_plan_search()` in Figure 9.

All the violation candidates are organized in a stack `viol_cand_stack`, which is the input argument of the recursive function. The function searches all the possible prediction plans recursively by examining the valid predictors of the violation candidates. Each search path enumerates the elements in the stack, computes the misspeculation cost of the valid predictor combinations of the popped violation candidates, and results with a prediction plan that records the prediction decision with each critical variable.

We use two bounding functions to effectively reduce the size of the searching space:

- `exceed_size ()`, which checks if the prediction code size exceeds the threshold *S* when a predictor is added into the prediction plan. If it is true, the predictor will not be used, and the search algorithm continues to check next available predictor or simply gives up predicting the variable;
- `exceed_cost()`, which checks if the misspeculation cost exceeds the threshold *C* when a violation candidate is decided not to be predicted. If it is true, the search path is stopped and the algorithm backtracks to other paths.

At first, all the violation candidates are stored in `viol_cand_stack` in the order according to their dependence relations, that is, if a violation candidate depends on some other ones, it will not be pushed in the stack before the other ones are pushed already. This ordering is achievable because there is no cyclic dependence between the violation candidates (The cyclic dependent violation candidates will be treated as one candidate by the compiler in the first place). This stack layout prunes lots of unnecessary searching paths early because a violation candidate can not be indirectly predicted if its dependent candidate is not predicted, and the decision as to dependent candidates is guaranteed to be made already according to the stack layout.

```
1: global var optimal_cost, optimal_plan;
2: predictor_plan_search( viol_cand_stack )
3: {
4:   path_end = TRUE;
5:   while( viol_cand_stack not empty ){
6:        viol_cand = viol_cand_stack.pop();

7:        for(each pred of viol_cand's predictors){
8:          pred_plan.add( pred );

9:          if( !exceed_size( pred_plan ) ){
10:              path_end = FALSE;
11:              predictor_plan_search(viol_cand_stack);
12:         }

13:         pred_plan.remove( pred );
14:       }//for
15:       if( exceed_cost( pred_plan ) )
16:         return;
17:  }//while

18:  if( path_end ){ //end of search path
19:     cost = misspec_cost(pred_plan);
20:     if( cost <= optimal_cost){
21:        optimal_cost = cost;
22:        optimal_plan = pred_plan;
23:     }
24:  }

25:  return;
26: }
```

Figure 9: *SVP prediction plan search algorithm*

The intermediate search result is kept in `pred_plan`, and the temporary optimal search result and its cost are kept in global variable `optimal_plan` and `optimal_cost`. We use a boolean variable `path_end` to indicate the end of a search path; and when meeting an end, i.e., a prediction plan is identified, the algorithm computes the misspeculation cost of the newly found plan, compares it with the temporary optimal result and chooses the smaller one to be the new optimal plan.

In the statement `7` of the pseudo-code, the compiler iterates through all the valid predictors of a violation candidate, including both the direct and the indirect predictors. For a critical variable `x`, its valid predictors are got basically by replacing the critical variables in its dependence slice one by one with their respective predictors. If the dependence slice is replaced completely with a prediction pattern, it is a direct-predictor; otherwise, it is an indirect-predictor. The real implementation is not so simple, but the idea keeps same.

6. Experiments and Results

We did experiments to evaluate the developed SVP technology with SPECint2000 benchmarks. The results with SPECint2000 benchmarks are encouraging, and demonstrate that software value prediction we developed is an effective and efficient technology to deliver TLP performance even without special value prediction hardware support.

6.1 Evaluation Methodology

We evaluated SVP with SPECint2000 benchmarks on our SPT simulator, which is an in-house data-flow IPF simulator developed for our SPT architecture and compiler research. 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. The architecture details can be found in [9]. Since our focus here is the SVP methodology and implementations, we didn't intend to explore and evaluate different architecture models for TLS in this paper. We used the current TLS model to demonstrate the SVP general methodology and effectiveness.

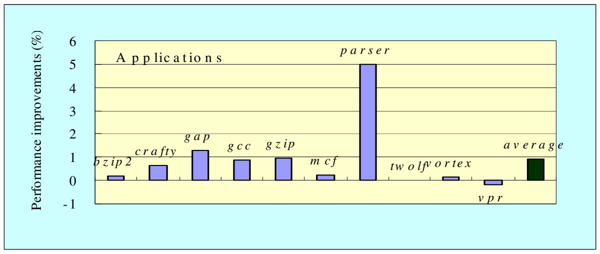
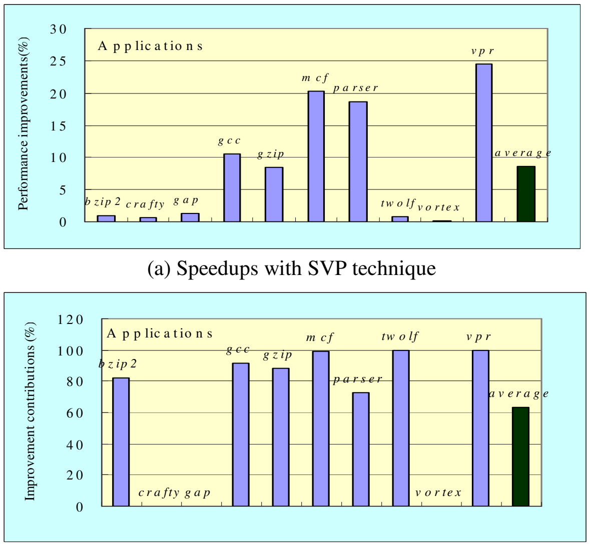


Figure 10*: Baseline SPT speedups without SVP*

Since SPT compiler has implemented a code reordering algorithm to transform SPT loops, we use that as base line to study the incremental effect of SVP. Many induction variables are handled by code reordering without requiring value prediction. Without SVP, our compiler will reorder code to minimize cross-iteration data dependences. Only in cases where code reordering cannot be applied to reduce dependences, SVP is applied.

The performance improvements achieved by code reordering are given in Figure 10. We can see that most applications get only marginal performance gains except *parser*, which has a close to 5.0% speedup; and the average improvement for all the ten applications is only 0.92%, almost negligible. *Vortex* is expected to have low speedup because it lacks appropriate loops for iteration-based speculative execution: The body size of its loops are too large for the processor store buffer to hold the temporary speculation results. It can be improved with region-based speculation, but is not this paper's focus.



(a) Speedups with SVP technique

(b) SVP's contributions in total speedups

Figure 11: *Speedups with SVP and its contributions*

6.2 TLP Performance with SVP

When SVP technology is applied, we get the speedups in Figure 11. As clearly shown in the Figure 11(a), the average performance improvement is boosted from previous 0.91% to current 8.63%; and there are more than four applications have more than 10% speedups.

The contributions of SVP technology in total speedups achieved by SPT compiler is shown in Figure 11(b). For six of the ten SPECint2000 applications, SVP contributes more than 80% to the final performance; and the average contribution for all applications is 63%! This clearly demonstrates the essential effects of value prediction in delivering TLP performance.

Although SVP can improve the TLP performance dramatically, there are still four applications except *vortex* showing less than 4% speedups. The reason is our SVP technology is not fully tuned in SPT compiler, so some important loops are not transformed. In order to understand the potential of SVP technology, we applied it manually with some of the remaining loops in SPECint2000 applications. The manual transformations in our study were applied because the current ORC compiler did not support the needed optimization or analysis. We could implement those optimizations and analyses if time allowed. For example, in order to best transform a loop in \"twolf\", the compiler needs to resolve memory aliasing across procedure boundaries. The current ORC does not provide that support. However, this can be done either by inlining the corresponding function or by type-based inter-procedural alias analysis. We were very careful in deciding what manual transformation that we could apply. We only apply those that are practically feasible. Figure 12 shows the resulted potential speedups.

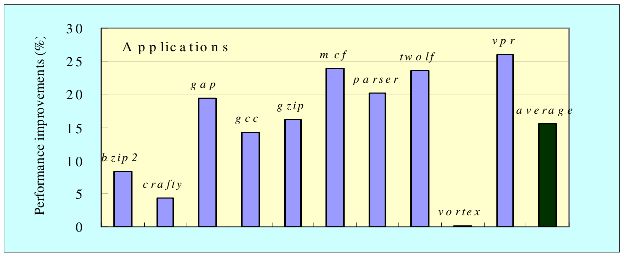


Figure 12*: Potential speedups with SVP technique*

In Figure 12, all the applications except *vortex* achieve higher than 4% performance improvement; and the average potential speedup can be more than 15% even with *vortex* counted.

6.3 SVP Runtime Overhead

SVP conducts the value prediction purely in software, so it could be interesting to see how much runtime overhead it incurs. We examine the overhead by running the SVP-transformed executables sequentially, and then compare the running time with that of non-SVP-transformed executables.

The result show that the runtime overhead of SVP is marginal and almost negligible for most of the applications. The average overhead is less than 1%, meaning SVP is efficient in accomplishing its goal. We found it is interesting that *gap* and *twolf* have even smaller running time with the SVP-transformed executable, which is contra-intuitive in the first glance but is actually reasonable, because we found the prediction code effectively prefetches some memory variables so that the cache misses have been reduced.

6.4 Contributions of Different Predictors

We want to know how different predictors are used in the applications, so we collect the contributions of different predictors to the total speedups. We classify the predictors into five categories.

- "const-stride" refers to the contribution is made by the loops that are transformed with only constant-stride predictors;
- "last-value + const-stride": with both last-value and constant-stride predictors, and without any other predictors;
- "indirect-predictor": with only indirect-predictors;
- "code-reordering": refers to the contribution is by SPT code reordering without prediction;
- "others": with other predictors that can not be classified into the above four categories.

The data in Figure 13 shows that, last-value and constant-stride predictors are mostly important in all predictors: They in combination (const-stride and last-value+const-stride) contribute more than 60% to the total speedups. Applications *parser* and *twolf* achieve their performance mainly through indirect-predictor, which is because the predicted variables in these two applications are mostly pointer-chasing variables.

The data also shows the importance of direct-predictor over indirect-predictor, meaning the critical variables largely have simple prediction patterns they can be identified by value profiling directly.

Code-reordering alone contributes only a minor portion, which again demonstrates the essential value of value prediction in achieving good TLP performance.

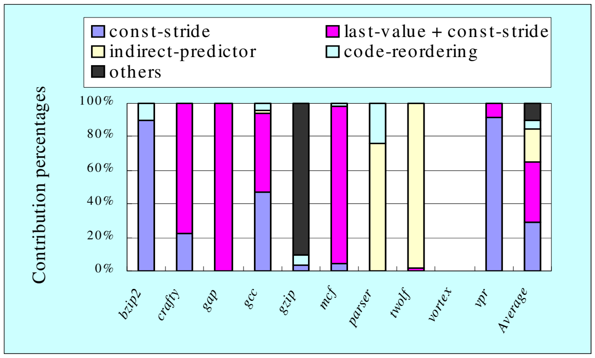


Figure 13*: Contributions of different predictors to total speedups*

6.5 Value Profiling Efficiency

Finally let us have a look at the value profiling efficiency. In our experiments, the average number of the profiled variables per loop is only 1.72, which is pretty small. The profiling time increases significantly with an average of 105.79 times of the original application execution time. One important reason is, we profiled lots of variables that are considered by the compiler as critical variables just because they are aliased with some critical variables, which can be improved by enhancing the alias analysis ability of the compiler. The profiling efficiency can further be improved by sample profiling.

6.6 Input Set Sensitivity

We used the "train" input sets for profiling. The results reported so far in the paper also used the "train" input set for evaluation. This corresponds to the ideal value prediction achievable by our SVP methodology and answers the ultimate performance question (i.e., on the best performance gain with SVP). We are fully aware of the input-set sensitivity problem which comes immediately after the performance question. So far we did have reasonable evidence that our results and conclusions are not sensitive to the input set used, though not strong enough.

we have run the same generated code but using the "ref" input sets and collected the performance results for seven Spec2000Int benchmark programs (Because of the time and resource issues, we are still collecting the rest ones.) Below in Table 1 is the comparison of the performance gains between those using "train" input sets and those using the "ref" input set. As expected, the performance gains using "ref" input sets are not as good as in the ideal case mostly. However, the differences are not large actually: the average speedup difference is only 7.79%, meaning the value predictability of the selected variables remains similar with different input sets.

Table 1: SVP speedup sensitivity to different input sets

	crafty	***gap***	***mcf***	***parser***	***twolf***	***vortex***	***vpr***	**average**
with train (%)	4.84	19.4	23.95	20.18	23.53	0.14	25.96	16.86
with ref (%)	3.69	19.4	19.71	16.96	25.81	0.14	23.1	15.54
diff (%)	23.76	0.00	17.70	15.96	-9.69	0.00	11.02	7.79

Basically our initial finding is, the different input sets do bring some differences in prediction accuracy, but not significantly. We studied the source code for the reason, and found most of the important predicted variables are either scalars or pointers.

The scalars are normally changed in a certain pattern (increment or bit-shift, for examples) under some conditions. If the condition is evaluated to be true in many iterations, the scalar can be predicted with the pattern; otherwise, it can be predicted with last-value pattern. Well under the two different input sets of our study, the condition evaluation result does not show big different trends. That means the same prediction pattern is applicable for both.

The pointer values are mainly decided by the memory management library which is the same for both input sets, so the prediction patterns of them are not changed much either.

We are not arguing the SVP technique is insensitive to different input sets, and we are looking for alternative approaches that can inherently eliminate the sensitivity issue.

7. CONCLUSIONS

Value prediction has been considered to be a valid approach to break data-flow parallelism limit. In this paper, we demonstrate that, value prediction is actually essential for the speculative thread-level parallelism, and can be achieved purely in software without any prediction hardware support. We show that, software value prediction can be achieved systematically and effectively by SVP compilation technology, which brings up to 15.63% performance improvement on average for SPECint2000 benchmarks with loop-based thread-level speculation. We also studied the contributions of different predictors, and found last-value and constant-stride predictors are most important for good speedups.

SVP compilation technology can be applied in areas beyond the loop speculation, such as region speculation, call-continuation speculation, etc. And it is not necessarily beneficial only to thread-level parallelism. For example as our data shown with *gap* application, SVP can effectively reduce cache misses so as to improve even the sequential execution performance. In SPT compiler, value profiling and SVP transformation are both carried in static compilation, we are also researching in dynamic profiling and just-in-time SVP transformation.

8. REFERENCES

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

[2] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao and Tin-Fook Ngai, "A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs", in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, Washington, D.C., June 9-11, 2004.

[3] 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.

[4] 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

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

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

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

[8] Shiwen Hu, Ravi Bhargava, and Lizy K. John, "The role of Return Value Prediction in Exploiting Speculative Method-Level Parallelism", The First Value-Prediction Workshop, San Diego, CA, June 7, 2003.

[9] Xiao-Feng Li, Zhao-Hui Du, Chen Yang, Chu-Cheow Lim, Qing-Yu Zhao, William Chen, Tin-Fook Ngai, "Speculative Parallel Threading Architecture and Compilation", 4th Workshop on Compile and Runtime Techniques for Parallel Computing, Oslo, Norway, June 2005.

[10] Xiao-Feng Li, Zhao-Hui Du, Qingyu Zhao, and Tin-Fook Ngai, "Software value prediction for speculative parallel threaded computations", The First Value-Prediction Workshop, San Diego, CA, June 7, 2003.

[11] 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.

[12] 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.

[13] 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.

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

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

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

[17] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. Proceedings of Intl Symp. On Computer Architecture 2000, pp. 1-24.

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

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

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

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

[22] 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.

[23] J. G. Steffan, C. B. Colohan, A. Zhai, T, C. Mowry. "Improving Value Communication for Thread-Level Speculation" , The Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), Boston, MA, USA, Feb 2002..

[24] M. Cintra, J. Torrellas, "Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors" The Eighth International Symposium on High-Performance Computer Architecture (HPCA'02), Boston, MA, USA, Feb 2002.