Speculative Parallelization Using State Separation and Multiple Value Prediction

Chen Tian, Min Feng, Rajiv Gupta
University of California, CSE Department, Riverside, CA, 92521
{tianc, mfeng, gupta}@cs.ucr.edu

Abstract

With the availability of chip multiprocessor (CMP) and simultaneous multithreading (SMT) machines, extracting thread level parallelism from a sequential program has become crucial for improving performance. However, many sequential programs cannot be easily parallelized due to the presence of dependences. To solve this problem, different solutions have been proposed. Some of them make the optimistic assumption that such dependences rarely manifest themselves at runtime. However, when this assumption is violated, the recovery causes very large overhead. Other approaches incur large synchronization or computation overhead when resolving the dependences. Consequently, for a loop with frequently arising cross-iteration dependences, previous techniques are not able to speed up the execution. In this paper we propose a compiler technique which uses state separation and multiple value prediction to speculatively parallelize loops in sequential programs that contain frequently arising cross-iteration dependences. The key idea is to generate multiple versions of a loop iteration based on multiple predictions of values of variables involved in cross-iteration dependences (i.e., live-in variables). These speculative versions and the preceding loop iteration are executed in separate memory states simultaneously. After the execution, if one of these versions is correct (i.e., its predicted values are found to be correct), then we merge its state and the state of the preceding iteration because the dependence between the two iterations is correctly resolved. The memory states of other incorrect versions are completely discarded. Based on this idea, we further propose a runtime adaptive scheme that not only gives a good performance but also achieves better CPU utilization. We conducted experiments on 10 benchmark programs on a real machine. The results show that our technique can achieve 1.7x speedup on average across all used benchmarks.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Performance, Languages, Design, Experimentation

Keywords Speculative Parallelization, Multicore Processors

1. Introduction

Extracting thread level parallelism from a sequential program is very important for improving performance on widely available multicore processors. Unfortunately, manual code parallelization by programmers is a time-consuming and error-prone process. Consequently, compiler based automatic parallelization techniques have drawn much attention of researchers. Many earlier works on DOALL parallelism [12, 16] focus on identifying loops without cross-iteration dependences. However, most sequential programs cannot be easily parallelized by a compiler due to the presence of cross-iteration dependences. To handle such dependences DOACROSS parallelism techniques [3, 18] use explicit communication to pass values between threads. Synchronization and send/receive instructions are inserted by the compiler to enforce cross-iteration dependences. A *receive* call blocks the recipient till the needed value becomes available. This blocking results in a degree of serialization. In addition, significant amount of time is spent on communication between different threads when considerable number of variables' values are communicated.

Thread level speculation (TLS) [5, 6, 11, 13–15, 21, 26–31, 36] is another approach that parallelizes sequential programs. In TLS techniques, multiple threads are created by the compiler to execute different parts of a sequential program (e.g., loop iterations) in parallel. An optimistic assumption that no dependence exists between the selected parts is made during compilation. Thus, TLS techniques can effectively exploit loop parallelism when the cross-iteration dependences are either absent or rarely manifest themselves at runtime. Note that TLS must be able to detect any misspeculations (i.e., dependence assumption violations) at runtime and appropriately deal with the speculative results to ensure the correctness of program execution. These functionalities of TLS can be implemented in hardware, software or a combination of both. However, if cross-iteration dependences of a sequential loop frequently take place at runtime, prior TLS techniques cannot improve the program performance. The reason is that these frequent dependences will cause the speculation to fail very often and thus wipe out the benefits of parallelism. One solution is to explicitly pass the value [35]. However, this approach will inherit the problems of the DOACROSS approach.

In this paper we propose a compiler technique, which uses state separation and multiple value prediction to speculatively parallelize loops in a sequential program that have frequently arising cross-iteration dependences. In our technique, we create multiple versions for a *later* loop iteration by using different predictions for the *live-in variables* (i.e., variables whose values must be obtained from an *earlier* iteration). These versions are executed in separate memory states. If one of these versions turns out to use correct predictions for all predicted variable values, then the performance is improved due to the parallel execution of the two iterations, i.e., the *earlier* iteration and the *later* iteration.

The remainder of this paper is organized as follows. Section 2 motivates our work and provides an overview of the state separation based speculative execution model. Section 3 describes how we generate multiple value predictions and how the code is transformed. A runtime adaptive scheme is proposed in section 4. Section 5 gives the evaluation results followed by discussion of related work in section 6. Conclusions are given in section 7.

2. Overview of Our Approach

2.1 Motivating Example

It has been observed that in some programs dependences may not occur frequently. To take advantage of such *infrequent dependences*, thread level speculation (TLS) based parallelization techniques have been proposed. The key idea of these techniques is to make the assumption that there is no dependence between the two sequential regions and execute them in parallel. However, if a dependence manifests itself during execution, then the speculation fails and the runtime system is required to detect and recover from misspeculation. Fig. 1 shows a speculative parallelization example where the loop iterations can be speculatively executed in parallel. Specifically, there is a cross-iteration dependence on variable *latest_config* between statements at lines 1 and 9. However, if the first condition *cond1* evaluates to false most of the time, then *latest_config* will not be updated frequently. Consequently, speculating on the absence of the dependence between line 1 and 9 is a good choice, as this speculation is frequently successful, which enables the parallelism to be aggressively exploited.

```
var1=... ;
var2=... ;
...
     while (...){
1        compute(..., latest_config);
2        if (cond1){
3            if (cond2){
4                x = var1;
5            }
6            else {
7                x = var2;
8            }
9            latest_config = config[x];
10       }
         ...
     }
...
```

Figure 1. Speculative Parallelization Example.

Unfortunately, if the condition *cond1* is always or frequently true at runtime, using speculative parallelization technique will not speed up the execution as misspeculation will occur frequently. The dependence carried by variable *latest_config* will cause the loop iterations to be executed sequentially. Moreover, the overhead of the technique such as isolating speculative states and dealing with misspeculation could make the performance even worse.

In this paper, we propose a technique, *multiple value predictions*, that can exploit the parallelism when such *frequent* dependences exist. The key idea is that for every two consecutive iterations that have data dependences on some variables (a.k.a *live-ins*), we predict the value of such variables for the second one. If the prediction is correct, then executing these two iterations in parallel using the predicted values for the later iteration will yield a speedup. For example, in Fig. 1, if both *cond1* and *cond2* are always true, then predicting *latest_config* to be *config[var1]* will enable speculative parallelization to succeed and lead to a better performance.

However, a single predicted value may not be very accurate. For instance, consider the scenario for Fig. 1 in which *cond1* and *cond2* keep evaluating alternately to true and false. Thus, a single prediction for the value of *latest_config* is not effective as it is not frequently successful. To solve this problem, we employ multiple predictions, each giving rise to a distinct version of the second iteration. The idea is that among all predictions that are chosen, it is highly likely that one prediction will turn out to be correct and the corresponding version will generate the correct result. More importantly, the correct result is computed in parallel with the execution of the first iteration. In other words, we will exploit parallelism by executing two consecutive iterations in parallel.

In the example shown in Fig. 1, for the second iteration of every pair of consecutive iterations, we can create three versions that are capable of generating the variable *latest_config* based on different path selections. Table 1 shows these versions. In the first version, we assume that both *cond1* and *cond2* are true in the first iteration and thus *latest_config* is set to *config[var1]* when the computation starts in the second iteration. In the second version only *cond2* is assumed to be false, and we set *latest_config* to be *config[var2]* at the beginning of the second iteration. Finally, in the third version we assume *cond1* is false in the first iteration. Thus, the computation of the second iteration can be directly started. For these three versions, one of them must be correct and thus lead to an execution speedup. This prediction method is essentially based upon collecting data slices of *latest_config* and creating one version for each distinct data slice. In Section 3, we will describe further details of our prediction method.

Version Number	Path In An Earlier Iteration	Prediction In A Later Iteration
1	$cond1$=true, $cond2$=true	x=$var1$; *latest_config*= *config[x]*;
2	$cond1$=true, $cond2$=false	x=$var2$; *latest_config*= *config[x]*;
3	$cond1$=false	No Prediction Code

Table 1. Three Versions For Generating *latest_config*.

2.2 State Separation Based Execution Model

To support multiple prediction schemes, we employ a state separation based execution model of [28, 29]. In particular, a program is compiled to contain one *main* thread and multiple *parallel* threads. For any two consecutive iterations, the first one is executed by the main thread and each version of the next one is executed by a parallel thread.

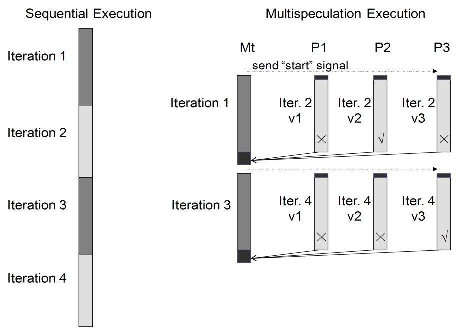


Figure 2. Thread Execution Model.

Fig. 2 shows the thread execution model. The original sequential execution, which consists of 4 iterations, is shown on the left. The corresponding parallel execution is shown on the right. As we can see, there is one main thread and multiple parallel threads. The main thread executes iteration 1 and 3 while parallel threads execute different versions of the iterations 2 and 4. A parallel thread does not begin execution until it receives the *start* signal from the main thread.

The dark region at the start of each parallel thread represents execution of code that predicts the values of the live-in variables that are to be computed by the previous iteration. Following this code, the computation of the current iteration is performed. The dark region at the end of the main thread's execution represents execution of code that validates the results. After finishing its own computation, the main thread needs to identify the parallel thread executing the correct version of the next iteration, and uses its result to continue the execution. The parallel thread that generates the correct result is also called the *winner*. In Fig. 2, we show that parallel threads P2 and P3 are the winners for iterations 2 and 4 respectively. In this execution model, every two dependent iterations can be executed in parallel and hence the theoretical speedup for the parallel execution is 2.

The key characteristic of our execution model is **state separation**, according to which the non-speculative state of the program maintained by the main thread is kept separate from the speculative state of the computation maintained by the parallel threads. To achieve state separation, we logically divide the entire shared memory space into three disjoint partitions (D, P, C). The D memory is the part of the address space that reflects the *non-speculative state* of the computation. Only the main thread can update the D space. The P memory is the part of the address space that reflects the *speculative state* of parallel threads. The results produced by the parallel threads are communicated to the main thread that then performs updates of D. The C memory is part of the address space that contains the coordinating state of the computation. The coordinating state is maintained to synchronize the actions of the main thread and the parallel threads and also to track how the speculatively-read values are copied or predicted so that misspeculation can be detected and the speculative results can be committed. Since the D and P memories are used by all threads, they must support **stack**, **global** and **heap** sections and each of these sections must provide state separation. The mechanisms used to provide state separation in each if these sections are described next.

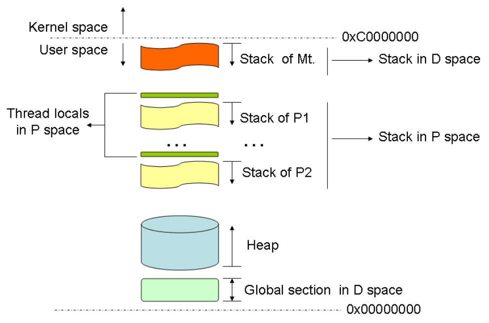


Figure 3. Separation Of Stack And Global Section.

Stack Separation. For a sequential program, all local variables are allocated on the stack and accessed through a stack pointer. When multiple threads co-exist, each of them has its own stack and stack pointer as shown in Fig. 3. Since our model is implemented using POSIX thread on Linux, this is automatically achieved by pthread library and OS. The default size of the stack allocated for each thread is 10M.

To avoid the stack overflow of each thread, a safety check needs to be performed when a stack grows. This is also automatically achieved by the OS. In particular, when the size of a stack exceeds its limit, OS sends a signal to the running process. In our execution model, this signal is captured by the process. Within the signal handler, we can reset the size of each parallel thread's stack by calling *pthread_attr_setstacksize()*.

Global Section Separation. Variables stored in the global section can only be used by the main thread. Parallel threads need to maintain speculative copies of these variables to achieve state separation. However, creating a copy of the entire global section for each parallel thread is wasteful because parallel threads do not run any more after they complete the speculative computations. Therefore, we only maintain one global section, which is used by the main thread. For every global variable used by a parallel thread, we create a local copy in the thread local storage above the parallel thread's stack as shown in Fig. 3. As a result, we do not need the overflow check on the global section because its size is not increased at runtime.

Heap Separation. Heap is used to support dynamic memory allocation in a sequential program. The allocation is performed through a memory allocator such as *malloc* library call. In our multithreading based execution model, however, we do not maintain a separate heap for each thread. This is because it is very hard to predict how much heap a thread will use at runtime. Thus, only one heap is used by the whole process. Logical separation is achieved as follows. When a heap chunk is allocated to the main thread, it is considered as D space heap. If it is allocated to a parallel thread, it is considered as P space heap. The safety check of heap access is simply done by checking the return value of the memory allocator. Specifically, if a *malloc* call fails in a parallel thread's execution, the parallel thread will free all the memory resources and exit.

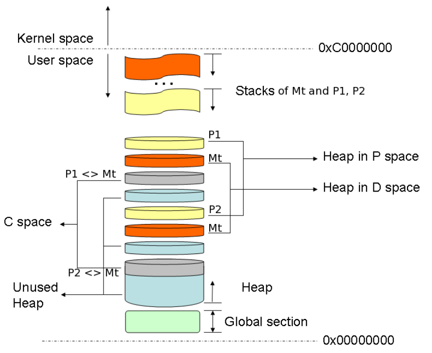


Figure 4. Separation Of Heap.

Besides the D space and P space that support the execution of different threads, we also allocate a buffer for each parallel thread to coordinate the execution of the parallel thread and the main thread. For example, if a variable (stack, global or heap) maintained by the main thread in D space is used by a parallel thread, a local copy of the variable is created in the corresponding P space. The mapping information of this variable (from a D space address to a P space address) needs to be stored in the buffer. C space is essentially the collection of all these buffers. In our model, the main thread allocates them by calling the *malloc* function. Thus, they are also on the heap at runtime. Each of them is deallocated when the corresponding parallel thread finishes its execution.

Fig. 4 illustrates how the heap of a process is used in our execution model. The figure shows that six heap chunks have been allocated. Two are allocated by the main thread, and thus considered in D space. Two chunks requested by parallel threads P1 and P2 respectively are considered in P space. Essentially, they are the duplicate copies of some D space heap chunk in the speculative state. Another two are allocated for coordinating the main thread and parallel threads, so they are logically in C space. The rest of heap is unused. The location of each memory chunk in the heap is decided by the memory allocator at runtime.

3. Basic Scheme of Multiple Value Predictions

3.1 Choosing Parallelization Candidate

Our technique is based on value predictions, and therefore resolves the situation where tasks of a program cannot be done in parallel due to dependences. In this work, we mainly focus on applying the technique to loops where each loop iteration is considered as a task. A loop is a good candidate if the following two conditions are satisfied:

- The loop has *frequent* loop carried dependences; and
- The values carried by loop dependences are *predictable*.

The first condition requires the examination of loop dependences. Although the dependence analysis can be performed using either static information (compiler based) or dynamic information (profile based), compiler based analysis does not work well for speculative parallelization. The reason is that it does not tell how often a dependence manifests itself at runtime. This frequency information is the key to selecting candidates for speculative parallelization. Therefore, profiling based loop dependence analysis is used for this work. In particular, the frequencies of cross-iteration dependences are captured during the profiling run. If the ratio of the number of iterations involving such a dependence to the total number of iterations is above a threshold, the dependence is considered as being frequent.

The second condition emphasizes that values of live-in variables are predictable. In this work, we consider a variable to be predictable if its value can be computed through a *small backward data slice* [1]. This can be checked by analyzing the trace of each iteration in the profiling run and computing dynamic slices from the traces. If a variable's slice is very large, we can further shrink the slice by applying some value prediction methods. More details are described in section 3.2.

The dependence frequency and predictability conditions must both be satisfied for the application of our technique. They both can be checked based on the information collected from the profiling run. Note that if the first condition is not satisfied, then the program is a good candidate for application of other speculative parallelization approaches [5, 11, 13–15, 28, 29].

3.2 Generating Multiple Versions

3.2.1 Using Data Slices and Control Flow Paths

To construct a speculative version of the second iteration, we need to insert the value prediction code of live-in variables before the original loop iteration code. The most accurate value prediction for a variable is to compute the value by executing its *full slice* extracted from the first iteration. However, the size of the full slice can be as large as the computation of the whole iteration. Using such a slice to obtain the value will be the same as executing the two iterations sequentially.

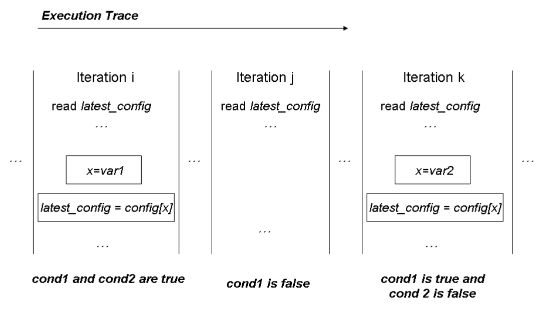


Figure 5. Trace Of Fig. 1.

To construct small prediction code and take advantage of the multiple value prediction model, following steps are used to generate multiple versions of the second iteration. First, we only compute the *backwards data slices* of a live-in variable. All the control dependences and the dependence chains of predicates are removed. Since different control flow paths may be taken in the first iteration, we can then compute the data slice on each *different path*. Consequently, we obtain multiple data slices for a live-in variable. At this point, we can create multiple versions for the second iteration based on different control flow paths taken by the first iteration. Specifically, each path corresponds to one version and the data slice on that path is used to predict the live-in variable. The data slice and path information are computed based on the profiling trace. Fig. 5 shows an example profiling trace of the *while* loop in Fig. 1.

In the example, we observe three things. First, there exists a frequent loop dependence on variable *latest_config* because two thirds of the iterations are involved in a dependence on variable *latest_config*. Second, there exist three different ways or data slices for computing this variable as shown in iterations i, j and k. All three are very small in comparison to the computation of the whole iteration. Therefore, we can create three versions for the second iteration. Last but not least, the path execution frequency, which equals to the frequency of the occurrence of each kind of slice, can be easily computed. This number will be used to compute the version confidence (VC) which reflects the probability of a version being correct.

$$VC = path\_frequency \times prediction\_confidence$$

The *prediction_confidence* for each live-in is always 1 if a data slice is used as the prediction code. If there are multiple live-ins, their data slices will be merged for the same control flow path. In other words, we construct one big data slice that is a union of all live-in variables' slices for each path and use it as the prediction code. The overall prediction confidence will be the product of individual live-in's confidence, which is still 1 in this case.

3.2.2 Reducing Data Slice on Each Path

Although by using data slices and path frequencies in generating multiple speculative versions we greatly increase the likelihood of covering the correct prediction of live-in variables, the performance can still be limited due to large sizes of the data slices. For example, if the variable *var1* and *var2* in Fig. 1 are computed within the loop, then the data slices of *latest_config* may be very large. Besides, merging the slices of multiple live-in variables can also lead to a large slice. Executing a large slice in each parallel thread can nullify the benefits of parallelism.

To tackle this problem, we reduce the data slice on each path by computing a *partial slice*, where the value of a live-in variable can be computed based on the predictions of other variables in the original slice. Given the data slice on each control flow path, we use the algorithm as shown in Fig. 6 to backwards traverse the slice and construct the prediction code of a live-in variable.

The idea behind this construction is that we search a point in a data slice where all variables in the slice can be either computed or predicted with high confidence using some simple value predictor. The search range is limited by a predefined value $size$ (line 2 and 13), which can be set to a fraction of total number instructions in the iteration.

In the algorithm, the $boundary$, implemented as a FIFO queue, stores those variables that need to be predicted to execute the statements stored in $partial\_slice$, which compute the remaining variables appearing in the slice. This boundary queue initially contains the live-in variable var (line 1), and is used to backwards traverse the slice in a breadth-first fashion. Specifically, in each search iteration, the first variable in the $boundary$ is popped out (line 9), and its definition in the slice is identified and stored in the $partial\_slice$ (line 10-11). After that, all source variables in the definition are pushed into the $boundary$ (line 12).

```
function compute_partial_slice(){
      $input$ = a data slice of $var$;
      $output$ = {}; //prediction code
      $partial\_slice$ = {};//partial slice of $var$
      $size$ = predefined maximum size of $output$;
      $i$ = 0;
      $OCBQ$ = 0;
      $boundary$ = a FIFO queue;

1.    $boundary$.enqueue($var$);
2.    while($i < size$) {
3.        $c$ = get_predictability($boundary$)
4.        if ($c > OCBQ$) {
5.            $ouput$ = prediction_stmt($boundary$) +
6.                  $partial\_slice$;
7.            $OCBQ$ = $c$;
8.        }
9.        $v$ = $boundary$.dequeue($var$);
10.       $stmt$ = the definition of $v$ in $input$;
11.       $partial\_slice$ += {$stmt$};
12.       $boundary$.enqueue(source variables in $stmt$};
13.       $i$++;
14.   }
15.   return $output$;
}

function get_predictability(){
      $input$ = a set of variables;
      $overall\_confidence$ = 1;

16.   for each $var$ in $input${
17.       $c_1$ = valuePredictor1_confidence($var$, trace);
18.       $c_2$ = valuePredictor2_confidence($var$, trace);
          ...
19.       $c_N$ = valuePredictorN_confidence($var$, trace);
20.       $var$.pred_flag = method $i$ whose confidence is the highest;
21.       $overall\_confidence$ *= max($c_1$,$c_2$,...,$c_N$);
22.   }
23.   return $overall\_confidence$ ;
}
```

Figure 6. Prediction Code Construction For var.

A function *get_predictability* is called every time the boundary queue changes (line 3). In this function, we look up the trace of the profiling run to find the values of every variable stored in the queue (also called *boundary variables*). Since each variable may have a different value in a different loop iteration, we make a value sequence for each variable by considering its value at the beginning of each iteration. Then we apply different value predictors on this sequence to compute a confidence number of the prediction (line 17-19). For every variable, we choose its prediction method by identifying the one with the highest confidence number. This prediction method is stored in the global flag *pred_flag* maintained for the variable (line 20). We multiply these highest numbers and store the result into $overall\_confidence$ (line 21). This product is used as the overall confidence of the boundary queue (OCBQ) as it indicates how good the combined prediction of all variables in the current boundary queue is. A boundary queue that has the highest OCBQ will be used to construct the prediction code, which basically contains the predictions of variables in the boundary and the statements in the $partial\_slice$ (line 3-6). The function *prediction_stmt* is called every time a higher OCBQ is found (line 4-8). When generating the prediction statements for each variable, it uses the best prediction method determined in function *get_predictability*.

To construct function *get_predictability*, we use three different value predictors, *last value predictor* [19], *stride predictor* [25] and *context predictor* [34]. In the *last value predictor*, the same value is assumed as being used again. Therefore, the last used value needs to be saved. The *stride predictor* assumes that two consecutive values of a variable have a constant difference (*stride*). Therefore, the most recent stride and values are used to predict the next value. The *context predictor* assumes that the most recent values are most likely to be used. Thus, it maintains a history of most recent values (normally 4) in a buffer. The prediction is made by referring to the buffer. The confidence of a method is defined as the percentage of the correct prediction.

If a version is constructed through a partial slice instead of a complete data slice, its VC is computed using OCBQ instead of 1.

$$VC = path\_frequency \times OCBQ$$

When more than one live-in variable is considered, we must apply this algorithm to all their data slices. The generated prediction code is merged if the data slices are on the same path. As a result, the VC is computed using the overall OCBQ of each path, which is the product of individual live-in's OCBQ.

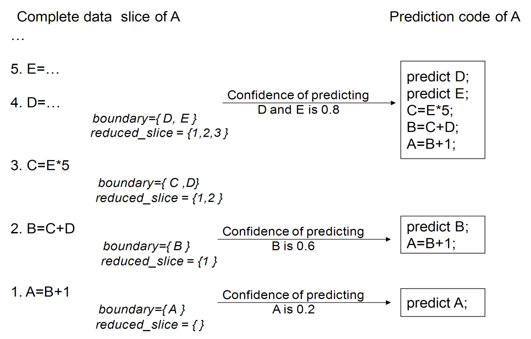


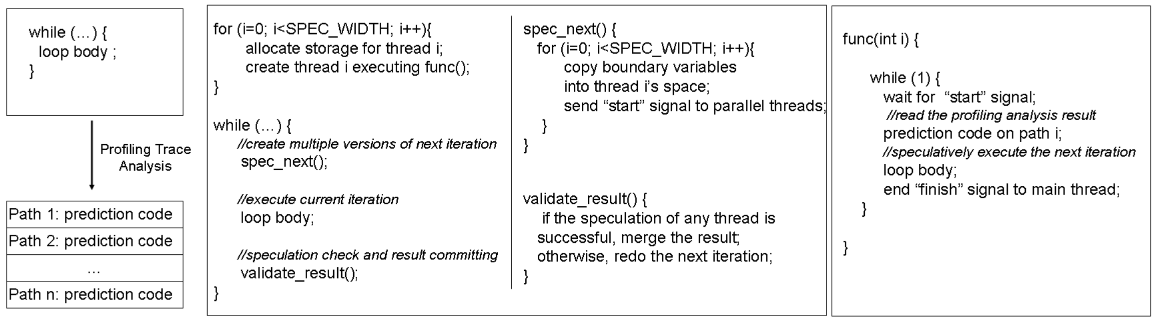
Figure 7. An Example Of Reduced Slice Construction.

Consider the example in Fig. 7. Suppose A is a live-in variable. On the left, we show the backward data slice of A on one control flow path. If the slice is very large, we can apply the algorithm to compute the prediction code for A. At the beginning, A is the only variable in the boundary queue. Since it can be predicted by one method with 0.2 confidence, A can be directly predicted and the prediction is put into the output (as shown in the bottom right). Then we continue to build better prediction code by examining the data slice backwards. According to the algorithm, we add the first statement $A = B + 1$ into the $partial\_slice$ set and recompute the boundary queue, which now contains B. Since the confidence of predicting B is 0.6 which is higher than 0.2, the prediction code for A will become the prediction of B and first statement. By continuing backwards traversal of the data slice we find the highest confidence when predicting both D and E. Therefore, we will get the best prediction code for A as shown on the top right, and the corresponding OCBQ is 0.8.

3.3 Code Transformation

Next we discuss the code transformation performed by our compiler - Fig. 8 shows the transformation. Given a loop as shown in Fig. 8(a), we first use trace-analysis tools to analyze the execution trace of the profiling run and generate the prediction code for live-in variables for different paths. The prediction code is generated by algorithm in Fig. 6. It is associated with a path represented by the branch history. The transformed parallel version contains the code for the non-speculative thread shown in Fig. 8(b) and for the speculative parallel threads shown in Fig. 8(c).

In Fig. 8(b) we can see that the main thread creates parallel threads before entering the loop. Each created thread executes the function *func* which contains a *while* loop waiting for the "start" signal so as to execute the loop body shown in Fig. 8(c).



```
while (...) {
  loop body ;
}
```

```
for (i=0; i<SPEC_WIDTH; i++){
      allocate storage for thread i;
      create thread i executing func();
}

while (...) {
   //create multiple versions of next iteration
   spec_next();

   //execute current iteration
   loop body;

   //speculation check and result committing
   validate_result();
}
```

```
spec_next() {
  for (i=0; i<SPEC_WIDTH; i++){
      copy boundary variables
      into thread i's space;
      send "start" signal to parallel threads;
  }
}

validate_result() {
   if the speculation of any thread is
   successful, merge the result;
   otherwise, redo the next iteration;
}
```

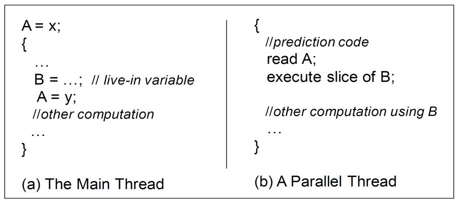
```
func(int i) {

  while (1) {
    wait for "start" signal;
    //read the profiling analysis result
    prediction code on path i;
    //speculatively execute the next iteration
    loop body;
    end "finish" signal to main thread;
  }

}
```

(a) Sequential Code. (b) Parallel Version - Main Thread. (c) Parallel Version - Parallel Threads.

Figure 8. Code Transformation.

After threads are created, the main thread enters the loop. It first creates multiple versions of the next iteration by executing *next_spec*. Then it executes the current iteration. Finally, it checks the speculation of each thread by executing *result_validation*.

Copying in the main thread. In function *next_spec*, the non-speculative thread needs to perform copying operations. In particular, the variables in the boundary set and modified in the loop body will be copied to parallel threads' space. This is important to avoid data races. Fig. 9 illustrates this with an example.



```
A = x;
{
  ...
  B = ...; // live-in variable
  A = y;
 //other computation
  ...
}
```

(a) The Main Thread

```
{
  //prediction code
  read A;
  execute slice of B;

  //other computation using B
  ...
}
```

(b) A Parallel Thread

Figure 9. An Example Of A Possible Race.

Assume B is a live-in variable and its backwards slice can be traced back to the statement $A = x$; (Fig. 9(a)). In other words, A is a boundary variable. To predict B, a parallel thread needs to read A and execute the slice of B as shown in Fig. 9(b). However, the main thread modified A after A is used for computing B. Hence, it is possible that the parallel thread reads the wrong value (y in the example) of A because of the race condition, and leads to a misprediction of B. To overcome this situation, the main thread should copy A for the parallel thread instead of allowing it to read A. After the copying operation, the main thread will send the "start" signal to parallel threads.

Copying in parallel threads. Parallel threads also need copying operations to ensure state separation. In particular, when a variable is about to be modified during the execution, it is copied from D space to P space. This is well known as the *copy-on-write* scheme. Similar to work [27–29], a mapping table is needed for each parallel thread to store the variables' mapping information. This is used when we merge the result.

Result validation. The result validation work is performed by the main thread. It needs to identify the correct version of the next iteration among all versions. To do that, the main thread needs to track the branch history of the current iteration and record the values of the boundary variables. When validating the result, the main thread simply needs to examine this information of the prediction code in each parallel thread. If a match is found, the corresponding parallel thread is the winner and its result will be merged into non-speculative state. Otherwise, the main thread has to re-execute the next iteration.

Committing results. The winner's results are committed by the main thread. Since the mapping table stored in C space contains the D space addresses of the modified variables, the main thread simply walks through the table and performs memory copying operations.

4. Adaptive Multiple Value Prediction Scheme

Although in the basic scheme the parallelism between every two consecutive iterations is exploited if one of the versions of the second iteration is correct, there are two problems with the basic scheme. First, it is possible that a small number of versions cover all popular execution paths, and thus the VCs of these versions are very high. Therefore, executing other versions, whose VCs are small, from the same iteration will waste cores. Second, the computation of each version's VC relies on the path frequency information of the profiling run. In a real run, different inputs may exhibit different path frequencies and thus change the VC. As a result, some versions may be less likely to be correct than expected causing cores to be wasted. Wasting cores can dramatically decrease the system throughput, if multiple applications co-exist. If the parallelized application runs alone, use of extra cores leads to waste of power.

To tackle this problem we propose an adaptive technique for better use of available cores. Our key idea is to consider the versions with a higher VC as candidates for executing additional iterations beyond the second iteration. Fig. 10 illustrates the idea.

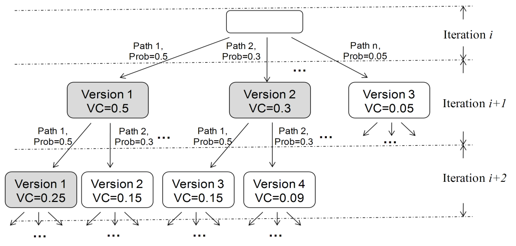


Figure 10. Selecting Versions With A Higher VC.

Suppose there are n different paths in a loop iteration. From the figure we can see that iteration *i+1* has n versions as denoted by v_1, v_2, ..., v_n, each of which corresponds to a certain path that might be taken in iteration i. The probability of each path being taken is marked on the edge. The VC of each version is shown in the node. When the first version of iteration *i+1* is executed, it still takes one of n paths in its own computation. As a result, the prediction code of iteration *i+2* has n^2 different cases leading to n^2 different versions. To calculate the VC of a version in iteration *i+2*, we need to multiply all the probabilities along the path back to the root (iteration i in the example). For example, the VC of the second version of iteration *i+2* is the product of 0.3 (the probability of iteration *i+1* taking path 2) and 0.5 (the probability of iteration i taking path 1).

Suppose P is the number of available cores for parallel threads. To assign the work to each parallel thread at runtime, the main thread identifies P versions that have the highest VCs as follows. It first calculates the VCs of all versions of iteration *i+1*. A version with the highest VC is then selected. Next, it computes the VCs of this version's children in iteration *i+2*. Assuming each iteration has n versions, it now has the VCs of *n-1* versions in iteration *i+1* and the VCs of n versions in the iteration *i+2*. It continues to select a version with the highest VC among these unselected versions and explores the children of the selected version. If two versions have the same VC, their parents' VCs are used to break the tie. Once the number of selected versions reaches P, the exploration terminates and the P versions are assigned to the parallel threads. In the example shown in Fig. 10, the versions represented by the shaded node will be selected if P is 3. This version-selection process in efficiently implemented using the maximum-heap data structure.

While we use extra cores to improve performance, we still need to avoid using cores to execute the versions that are unlikely to be a winner. This is important for achieving fairness among multiple applications being executed. Therefore, a threshold number is used to prevent a version with small VC from being executed. If the main thread cannot find P versions with VCs larger than the threshold, then the extra parallel threads are set to be inactive so that OS can schedule other applications on the remaining cores.

5. Experimental Results

5.1 Experimental Setup

Implementation. We implemented our technique using the Pin [20] instrumentation framework and the LLVM compiler infrastructure [17]. Fig. 11 shows the procedure for parallelizing a sequential program. We first compile a sequential program into its executable

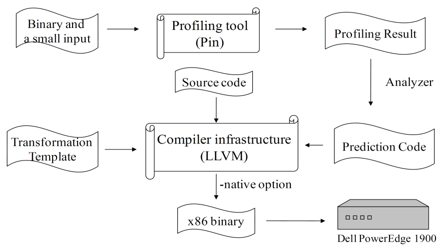


Figure 11. Experimental Framework.

with debugging information. Then we use our profiling tool to collect runtime information for outermost loops under a small input – the profiler is implemented by instrumenting the executable using Pin. The information collected includes dependences, values of variables at start of each iteration, and control flow paths taken with their execution frequencies. The dependences are classified into intra-iteration and cross-iteration dependences. The live-in variables are identified by looking at the cross-iteration dependences. The prediction code of these variables on each path is identified using intra-iteration dependences, value sequences, and control flow paths based on the algorithm shown in Fig. 6. Then, LLVM uses these predictions and a transformation template to recompile the sequential program into the parallelized version. Finally, we run the parallelized program with a larger input and collect the data under CentOS 4 OS running on a dual quad-core Xeon machine with 16 GB memory. Each core runs at 3.0 GHz.

Benchmarks. In our experiment, we use ten programs. Five of them, namely *dry*, *fldry*, *llu*, *mechcall* and *objinst*, are from the benchmark suite distributed with LLVM. Another five programs are from the SPEC2000 suite: *164.gzip*, *175.VCR*, *255.vortex*, *253.perlbmk* and *300.twolf*.

5.2 Performance Analysis

5.2.1 Performance of Basic Scheme

Fig. 12 shows the performance of each program when we apply the basic scheme where for every two consecutive iterations, different number of versions are created for the second one. It should be noted that throughout our experiments, the maximum number of speculative versions (threads) we allowed is 7 since the machine has 8 cores and we reserve one core for the main thread. To avoid thread idling, we unroll 10-20 iterations for the programs from LLVM [28, 29]. The degree of unrolling is chosen based on the tuning result obtained in the experiments. For the SPEC programs, no unrolling is needed because the loop body is large enough.

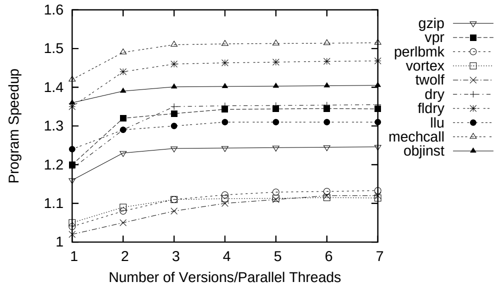


Figure 12. Overall Performance.

As we can see, the speedup for all benchmarks increases faster at the beginning and slower as more versions are used. The highest speedups ranging from 1.12x to 1.51x are achieved when three or four versions are used. This can be explained by Fig. 13 which shows the cumulative speculation success rate of each benchmark when increasing number of versions are executed in parallel. In Fig. 13 , we notice that using the first 3 to 4 versions significantly increases the success rate and thus leads to a big increment in the speedup for every program. When more versions are used, the total speculation success rate does not significantly increase any more, so these later versions provide little contribution to the performance. This is primarily because the paths corresponding to the later versions have low execution frequencies. In these programs, three or four hot paths are taken with over 95% probability. Consequently, the versions corresponding to the infrequently taken paths are not likely to be winners. In the case of *mechcall* and *objinst*, the control flow graphs are very simple and there are only three ways of computing live-in variables. Therefore, using three versions (threads) is enough to execute two consecutive iterations in parallel and the other threads remain idle.

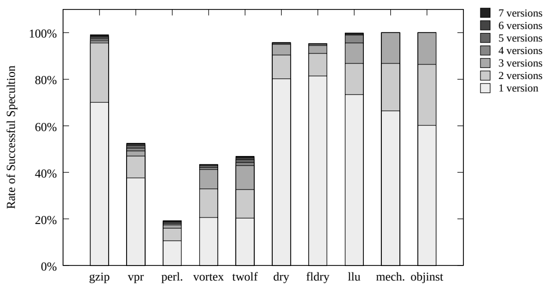


Figure 13. Speculation Success Rate.

From Fig. 13, we also observe that all SPEC benchmarks except for *gzip* have lower speculation success rates (less than 60%) than the other five LLVM benchmarks even when 7 versions are executed. This is because the data slices of *live-in* variables in the LLVM programs are small and hence the speculation success rates

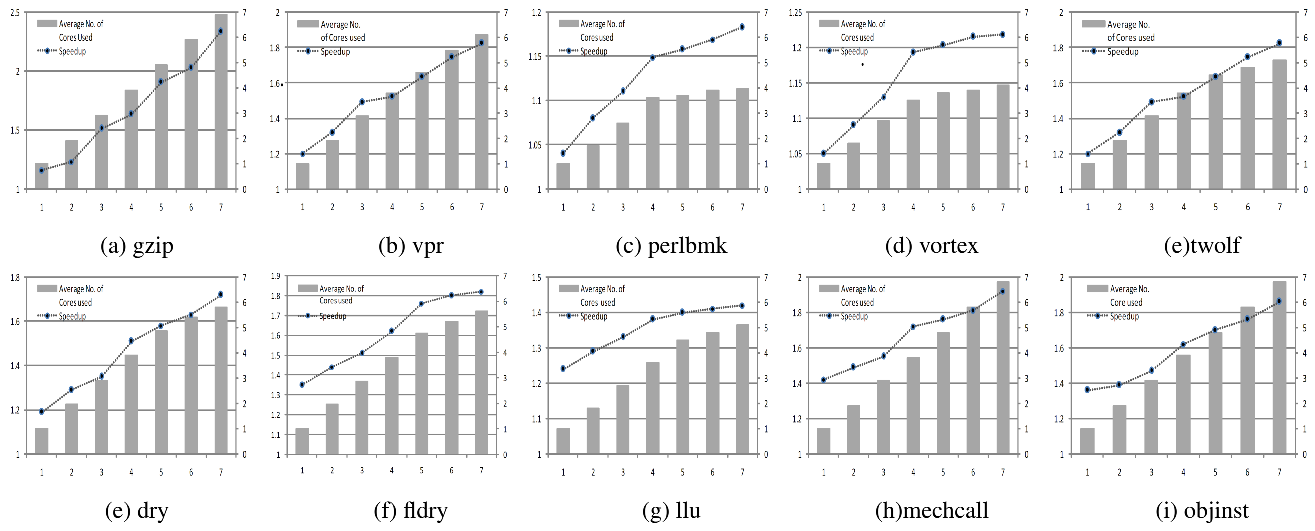


Figure 14. Performance Of Adaptive Scheme.

are only determined by the path coverage of the speculative versions. For most SPEC benchmarks, however, the value mispredictions on hot paths also causes the speculation to fail, and thus reduces the success rate. In the case of *gzip*, the value predictions of the live-in variables are very accurate and hence we observe a very high speculation success rate.

5.2.2 Performance of Adaptive Scheme

We also conducted a set of experiments for each benchmark when the adaptive scheme is used. Fig. 14 shows the results. For each program, we measured two numbers when different number of parallel threads are allowed. The first number is the speedup demonstrated by the line with markers. The left y-axis shows the speedup values and the x-axis shows the maximum numbers of parallel threads that are allowed. Compared to the basic scheme, additional speedups are achieved when more threads are used. The highest speedups range from 1.18x (*perlbmk*) to 2.33x (*gzip*) for all benchmarks. These numbers are achieved when 7 versions (parallel threads) are allowed to be used. This is because the adaptive scheme executes a few versions of the third or even fourth iteration at runtime. Since these versions are more likely to be correct than some versions in the second iteration, executing them allows us to exploit more parallelism and achieve higher speedups.

The second number we measured is the average utilization of cores. As described in section 4, to avoid wasting CPU resources, versions with very small VCs (less than 0.05 in our experiments) are not executed and the remaining threads are set to inactive. To measure the actual CPU utilization of each parallelized program, we recorded the number of versions that are selected every time by the main thread. The selected versions are the ones whose VCs are not only above a threshold number, but also among the P highest ones, where P is the maximum number of parallel threads allowed (shown on the x-axis). Then we compute the average of all these numbers obtained throughout the whole execution. The data are represented by the bars in each figure and the right y-axis shows the values. For all benchmarks, the utilization is very high when the maximum number of parallel threads allowed is less than 4. After that, some programs may not always use all cores to execute the speculative versions. In particular, when 7 threads are allowed, for *vortex* and *perlbmk*, the best speedup is achieved by only using around 4 cores on average. In other words, the main thread normally cannot find more than 4 versions with VCs above the threshold at runtime and hence saves the extra 3 cores without sacrificing performance. This saving for *llu* and *twolf*, is about 2 cores, and for *vpr*, *dry* and *fldry* is one core on average. In the case of *gzip*, *objinst* and *mechcall*, the parallelized program often uses all cores to exploit the parallelism.

5.2.3 Performance Comparison with Other Techniques

We compare the effectiveness of our technique with three other techniques. The first is DOACROSS [3, 18] where the values of all live-in variables are passed through explicit messages. In our experiments, the message-passing scheme is implemented through POSIX pipe which supports send/receive calls. The second technique is TLS, which optimistically assumes no cross-iteration dependences exist. Since our technique is implemented purely in software, we used our CorD implementation [28] for comparison. The last technique in our comparison is Mitosis [23] where all live-in variables are pre-computed through full slices. As mentioned earlier, the slice size is normally large. Therefore, an optimization has been proposed in [23] where the data slice on the path that is most frequently taken is kept. Note that Mitosis requires architectural support for speculative execution. For a fair comparison, we evaluate Mitosis using our software speculation approach. Specifically, we simulate Mitosis by generating one version for each iteration and constructing the pre-computation code by using the complete data slice on a path with the highest frequency.

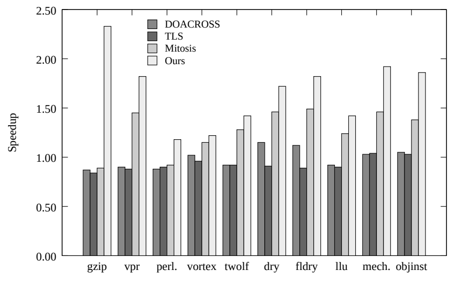


Figure 15. Performance Comparison With Other Techniques.

Fig. 15 shows the results of the comparison where 7 parallel threads are allowed for all techniques. From the figure, we can see that our multiple-value-prediction technique outperforms the other three techniques for all benchmarks. In most cases, DOACROSS and TLS slow down instead of speeding up sequential executions. The reason for DOACROSS is that it spends significant time on waiting for the values of live-ins, especially when such variables are used very early by a speculative thread. In the case of TLS, being too optimistic leads to excessive misspeculations.

Mitosis performs much better than DOACROSS and TLS for most programs. Compared to TLS, Mitosis has lower misspeculation rate because it uses full data slice to calculate live-ins. The reason for Mitosis not being as good as our technique is that it only uses one version of each iteration. Therefore, if more than one hot path exists, which is true for most benchmarks as indicated in Fig. 13, the misspeculation rate is high because the speculation on later iterations is prone to be wrong. As a result, the parallelization benefit is diminished. For *gzip* and *perlbmk*, Mitosis slows down the execution because in these two programs, the size of live-in variables' slices on the most frequent paths is very large. Without value predictions, Mitosis executes the loop almost sequentially and the overhead of the runtime system further degrades the performance.

5.3 Overhead Analysis

5.3.1 Time Overhead

Our execution model imposes the overhead on the execution of the parallelized program. We measured this overhead by breaking down the execution time into different categories for the parallel threads and the main thread respectively.

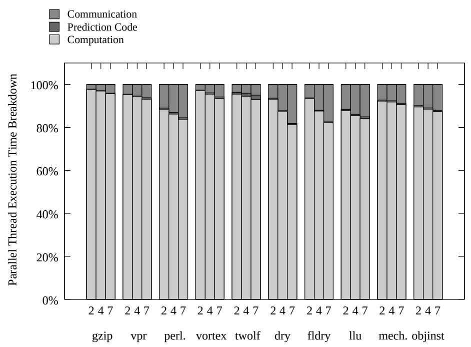


Figure 16. Time Breakdown: Parallel Threads.

Fig. 16 shows the average time breakdown of one parallel thread in case of using a total of 2, 4, and 7 parallel threads. As we can see, each parallel thread spent most time on the computation and less than 1% time on executing the prediction code. The rest of the time is spent on communication with the main thread. According to the results, the communication overhead rises as the number of threads increases. This is because the main thread controls the parallel threads by sending the start signal and examining the results sequentially and thus using more parallel threads leads to a longer waiting time for each.

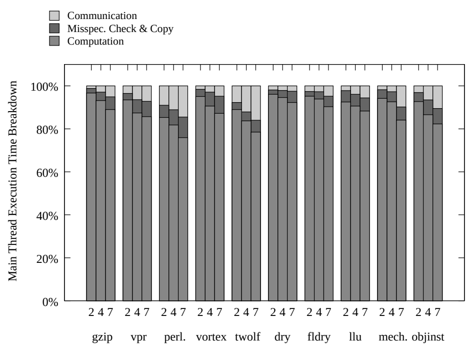


Figure 17. Time Breakdown: Main Thread.

Fig. 17 shows the breakdown of time for the main thread that is responsible for executing the sequential part and some iterations of the parallelizable loops, communicating with the parallel threads, and performing misspeculation checks and copying operations. From the figure, we can see that the *computation* category dominates the execution time for all programs. The fraction of time spent on communication, misspeculation check, and copying operations increases when more parallel threads are used. The sum of these two fractions, which reflects the overhead imposed by the execution model, is less than 25%. Considering the speedups we obtained, the benefit of exploiting parallelism outweighs the cost of implementation overhead.

5.3.2 Space Overhead

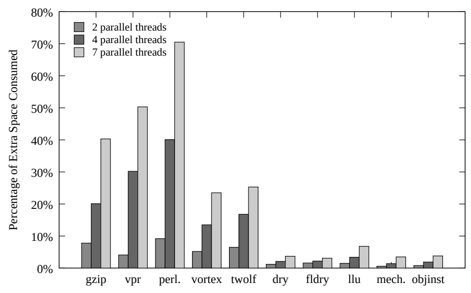


Figure 18. Space Overhead.

We also measures the space overhead incurred to obtain speedups by monitoring the peak value of memory consumption while the parallelized program is run with varying number of threads. As we can see from Fig. 18, when more threads are used, more space is used for all benchmarks. We notice that the parallelized SPEC benchmarks consume more memory especially when 7 threads are used (20%-80%). On the other hand, the parallel versions of the other 5 programs consume less than 7% extra memory regardless of the number of parallel threads. This is because SPEC programs are much larger and have more variables being used in each loop iteration. Consequently, each parallel thread has to maintain a copy of these variables which requires more memory.

6. Related work

DOALL/DOACROSS. To improve performance through loop parallelization, DOALL techniques were proposed decades ago [12, 16]. Due to the cross-iteration dependences, many programs cannot be simply parallelized. To solve this problem, DOACROSS technique was proposed [3, 18] which uses explicit send/receive calls or instructions to synchronize and exchange values between threads. However, the blocking at receives serializes the execution. As a result, DOACROSS techniques cannot greatly improve the performance, especially when the value of a live-in variable is used at the beginning of a thread's execution.

TLS techniques. Numerous thread level speculation (TLS) techniques have been proposed to aggressively exploit potential parallelism from a sequential program. Among these techniques, several are software based [4, 5, 9, 11, 13–15, 24, 28, 29]. Compared to hardware based solutions, software TLS does not require any non-trivial architectural modifications like special buffer [10, 22], versioning cache [8] or versioning memory [7].

While these software TLS techniques can be applied in array-only applications [4, 9, 24], pointer-based irregular applications [13–15], or general applications [5, 11, 28, 29], they are only effective in parallelizing a loop that does not have frequent cross-iteration dependences. Otherwise, excessive misspeculations occur.

Pre-computation technique. Quinones et al. [23] proposed the Mitosis compiler where the values of live-in variables are pre-computed through data slice on the most frequently taken path. Although this approach is effective in dealing with frequent cross-iteration dependences, it has two drawbacks. First, the data slice of one or more live-in variables on one particular path can be very large (e.g., *gzip*). Without value predictions, the pre-computation takes almost the same amount of time as executing one iteration in such cases. Second, the most frequently taken path is decided by the compiler using profiling results. However, there may exist more than one hot path in a loop execution at runtime. Moreover, the inputs used in the real runs are different from those used in the profiling runs. Therefore, the hot paths in the profiling runs may not be frequently taken in the real runs. Thus, picking one hot path at compile time may cause many misspeculations at runtime. Apart from these two drawbacks, Mitosis compiler does not fully support speculation. It relies on the hardware to detect misspeculations and handle speculative results, and hence is not a purely software speculation technique, but rather a hybrid one.

Multipath execution techniques. Using control flow paths to create multiple executions has also been used in architectural designs [2, 32, 33]. In these works, spare hardware contexts are used to execute instructions along the paths corresponding to different predictions of hard-to-predict branches. If one of these redundant executions is correct, then the penalty of the branch misprediction is greatly reduced. Since these techniques focus on improving performance through hardware changes, they are different from our software based compiler technique.

7. Conclusions

In this paper we presented a speculative parallelization technique that is implemented purely in software. By using multiple value predictions and state separation, this technique resolves frequent cross-iteration dependences and exploit the parallelism between consecutive loop iterations. The experimental results show that, on an average, our technique achieves 1.7x speedup across ten benchmarks.

Acknowledgments This work is supported by NSF grants CCF-0963996, CCF-0905509, CNS-0751961, and CNS-0810906 to the University of California, Riverside.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90*, pages 246–256.

[2] P. S. Ahuja, K. Skadron, M. Martonosi, and D. W. Clark. Multipath execution: Opportunities and limits. In *Supercomputing'98*, pages 101–108.

[3] M. G. Burke and R. K. Cytron. Interprocedural dependence analysis and parallelization. In *PLDI '86*, pages 162–175.

[4] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03*, pages 13–24.

[5] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07*, pages 1–12.

[6] M. Franklin and G. S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, 1996.

[7] M. J. Garzaran, M. Prvulovic, and J. M. Llaberia. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *TACO*, 2(3):247–279, 2005.

[8] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *HPCA '98*, pages 195–205.

[9] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98*, pages 1–12.

[10] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS '98*, pages 58–69.

[11] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *CGO '09*, pages 157–168.

[12] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[13] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09*, pages 3–14.

[14] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS '08*, pages 233–243.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07*, pages 211–222.

[16] L. Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, 1974.

[17] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88.

[18] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Comput.*, 24(3-4):445–475, 1998.

[19] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS '96*, pages 138–147.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200.

[21] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *In Supercomputing '99*, pages 365–372.

[22] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *ISCA '01*, pages 204–215.

[23] C. G. Quiñones, C. Madriles, F. J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05*, pages 313–325.

[24] L. Rauchwerger and D. A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[25] Y. Sazeides, Y. Sazeides, J. E. Smith, and J. E. Smith. The predictability of data values. In *MICRO '97*, pages 248–258.

[26] G. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95*, pages 414–425.

[27] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI '10*.

[28] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08*, pages 330–341.

[29] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Speculative parallelization of sequential loops on multicores. *International Journal of Parallel Programming*, 37(5):508–535, 2009.

[30] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[31] D. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95*, pages 392–403.

[32] A. K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: an optimal form of speculative execution. In *MICRO '95*, pages 313–325.

[33] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *ISCA '98*, pages 238–249.

[34] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *MICRO '97*, pages 281–290.

[35] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS '02*, pages 171–183.

[36] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *HPCA '99*, pages 135–141.