Copy Or Discard Execution Model For Speculative Parallelization On Multicores

Chen Tian Min Feng Vijay Nagarajan Rajiv Gupta
Department of Computer Science and Engineering
University of California at Riverside, CA, U.S.A
{tianc, mfeng, vijay, gupta}@cs.ucr.edu

Abstract

The advent of multicores presents a promising opportunity for speeding up sequential programs via profile-based speculative parallelization of these programs. In this paper we present a novel solution for efficiently supporting software speculation on multicore processors. We propose the Copy or Discard (CorD) execution model in which the state of speculative parallel threads is maintained separately from the non-speculative computation state. If speculation is successful, the results of the speculative computation are committed by copying them into the non-speculative state. If misspeculation is detected, no costly state recovery mechanisms are needed as the speculative state can be simply discarded. Optimizations are proposed to reduce the cost of data copying between non-speculative and speculative state. A lightweight mechanism that maintains version numbers for non-speculative data values enables misspeculation detection. We also present an algorithm for profile-based speculative parallelization that is effective in extracting parallelism from sequential programs. Our experiments show that the combination of CorD and our speculative parallelization algorithm achieves speedups ranging from 3.7 to 7.8 on a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors.

Keywords - multicores, speculative parallelization, copy or discard execution model.

1. Introduction

The advent of multicores presents a promising opportunity for speeding up sequential programs via profile-based speculative parallelization of these programs. The success of speculative parallelization is dependent upon the following factors: the efficiency with which success or failure of speculation can be ascertained; following the determination of success or failure of speculative execution, the efficiency with which the state of the program can be updated or restored; and finally, the effectiveness of the speculative parallelization technique, which is determined by its ability to exploit parallelism in a wide range of sequential programs and the rate at which speculation is successful.

In this paper we develop a novel execution model, named *Copy or Discard* (CorD), that effectively addresses the aforementioned issues. The key features of this execution model and our speculative parallelization algorithm are:

(Speculation and Misspeculation Detection) Under our execution model, a parallelized application consists of the *main thread* that maintains the non-speculative state of the computation and multiple *parallel threads* that execute parts of the computation using speculatively-read operand values from non-speculative state, thereby producing the speculative state of the computation. The main thread commits the speculative state generated by parallel threads *in order*; i.e., a parallel thread that is assigned an earlier portion of a computation from a sequential program must commit its results before a parallel thread that is assigned a later portion of a computation from the sequential program. Before committing results, the main thread confirms that the speculatively-read values conform to the sequential program semantics using version numbers.

(Maintaining State Via Copy or Discard) The non-speculative state (i.e., state of the main thread) is maintained separately from the speculative state (i.e., state of the parallel threads). After a parallel thread has completed a speculative computation, the main thread uses the *Copy or Discard* (CorD) mechanism to commit these results. In particular, if speculation is successful, the results of the speculative computation are committed by *copying* them into the non-speculative state. If misspeculation is detected, no costly state recovery mechanisms are needed as the speculative state can be simply *discarded*. Aggressive optimizations are proposed to minimize the cost of the *copying* operations.

(Speculative Parallelization Algorithm) We also present an algorithm for profile-based speculative parallelization that is effective in extracting parallelism from loops in sequential programs. In particular, a loop iteration is partitioned into three sections – the prologue, the speculative body, and the epilogue. While the prologue and epilogue contain statements that are dependent on statements in corresponding sections of the preceding iteration, the body contains statements that are extremely unlikely to be dependent on the statements in the body of the preceding iteration. Thus, speedups can be obtained by speculatively executing the body sections from different loop iterations in parallel on different cores.

Our experiments show that the combination of CorD and our speculative parallelization algorithm when applied to several sequential applications achieves speedups ranging from 3.7 to 7.8 on a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz and 16 GB of RAM. Our experiments also demonstrate that our optimizations for minimizing copying are highly effective.

2. Speculative Parallel Execution Model

2.1. Thread Execution Model

A parallelized application consists of the *main thread* that maintains the non-speculative state of the computation and multiple *parallel threads* that execute parts of the computation using speculatively-read operand values from non-speculative state, thereby producing the speculative state of the computation.

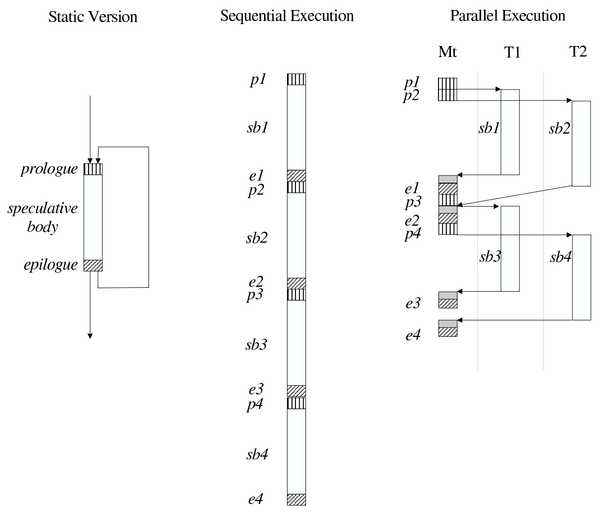


Figure 1. Thread Execution Model.

(Speculative Execution) Fig. 1 shows how threads are created to extract and exploit parallelism from a loop. We divide the loop iteration into three sections: the prologue, the speculative body, and the epilogue. While the prologue and epilogue contain statements that are dependent on statements in corresponding sections of the preceding iteration, the body contains statements that are extremely unlikely to be dependent on the statements in the corresponding section of the preceding iteration. Thus, parallelization is performed such that the main thread (Mt) *non-speculatively* executes the prologues and epilogues, while the parallel threads (T1 and T2 in Fig. 1) are created to *speculatively* execute the bodies of the iterations on separate cores. Speculative execution entails optimistically reading operand values from non-speculative state and using them in the execution of speculative bodies. The partitioning of a loop iteration, and the corresponding parallelization being employed, is based upon our analysis of several benchmarks.

(In-order Commit) Once a parallel thread has completed the execution of an iteration assigned to it, the speculatively computed results are returned to the main thread. The main thread is responsible for committing these results to the non-speculative state. The main thread commits the speculative state generated by parallel threads *in-order*; that is, a parallel thread that is assigned the speculative body of an earlier iteration must commit its results to the non-speculative state before a parallel thread that is assigned a later iteration commits its results.

(Misspeculation Detection) Before committing speculatively-computed results to non-speculative state, the main thread confirms that the speculatively-read values are consistent with the sequential semantics of the program. The main thread maintains version numbers for variable values to make this determination. In particular, if the version number of a speculatively-read operand value used during loop iteration *i* has not changed from the time it was read until the time at which the results of iteration *i* are committed, then the speculation is successful. However, if the version has been changed by an earlier loop iteration being executed in parallel on another core, then we conclude that *misspeculation* has occurred and the results must be recomputed.

2.2. Maintaining Memory State

The non-speculative state (i.e., state of the main thread) is maintained separately from the speculative state (i.e., state of the parallel threads). After a parallel thread has completed a speculative computation, the main thread uses the *Copy or Discard* (CorD) mechanism to commit these results. In particular, if speculation is successful, the results of the speculative computation are committed by *copying* them into the non-speculative state. If misspeculation is detected, no costly state recovery mechanisms are needed as the speculative state can be simply *discarded*.

To implement CorD, it is essential that thread isolation be provided such that no updates of the non-speculative state can be performed by the parallel threads. Once a parallel thread completes the execution of a speculative iteration body, it sends its results to the main thread which is responsible for updating the non-speculative state. To ensure thread isolation, the shared memory space is divided into three disjoint partitions $<$ D, P, C$>$ such that each partition contains a distinct type of program state (see Fig. 2).

(Non-speculative State) - D memory is the part of the address space that reflects the non-speculative state of the computation. Only the main computation thread Mt performs updates of D. If the program is executed sequentially, the main thread Mt performs the entire computation using D.

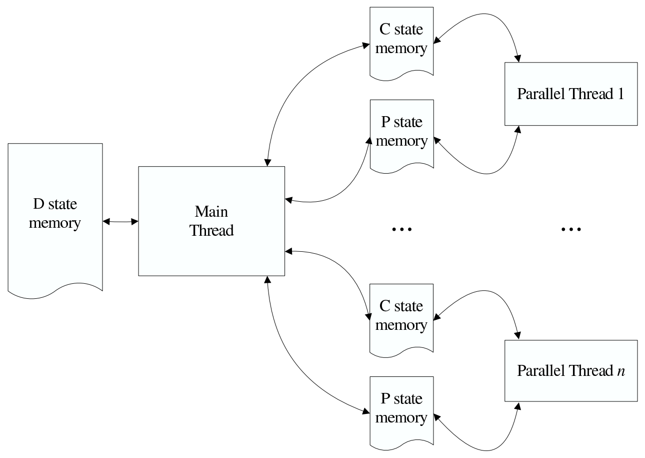


Figure 2. Maintaining Memory State.

If parallel threads are used, then the main thread Mt is responsible for updating D according to the results produced by the parallel threads.

(Parallel or Speculative State) - P memory is the part of the address space that reflects the parallel computation state, i.e. the state of the parallel threads Ts created by the main thread Mt to boost performance. Since parallel threads perform speculative computations, speculative state that exists is at all times contained in P memory. The results produced by the parallel threads are communicated to the main thread Mt that then performs updates of D.

(Coordinating State) - C is the part of the address space that contains the coordinating state of the computation. Since the execution of Mt is isolated from the execution of parallel threads Ts, mechanisms are needed via which the threads can coordinate their actions. The coordinating state provides memory where all state needed for coordinating actions is maintained. The coordinating state is maintained for three purposes: to synchronize the actions of the main thread and the parallel threads; to track the version numbers of speculatively-read values so that misspeculation can be detected; and to buffer speculative results computed by a parallel thread before they can be sent to the main thread to be committed.

When a parallel thread is created, both C state and P state memory is allocated for its execution. The speculatively-read operand values are copied from non-speculative D state to P state memory allocated to the thread. The thread executes, speculatively computing results into the P state memory. These results are communicated to the main thread for committing to the D state memory. Thus, during the execution of a parallel thread the state of the main thread in D state is isolated from the actions of the parallel thread. The C state memory allocated to the thread is used, among other things, by the main and parallel threads to signal each other when various actions can be performed.

2.2.1. Coordinating State for Realizing *CorD*. When the *speculative bodies* of two consecutive iterations of a loop in a sequential program are executed in parallel on two cores, the execution of the later iteration must be carried out speculatively under the assumption that the earlier iteration does not produce any value that is needed for the execution of the later iteration. The values of variables referenced by the later iteration are speculatively copied from D state memory to the P state memory, and using these copied values, the later iteration is speculatively executed. Before committing the results of the later iteration to D state memory, the main thread must check to see if the speculation was successful. If the earlier iteration has not modified the values of variables that were speculatively copied and used by the later iteration, then the speculation of the later iteration is successful; otherwise a misspeculation occurs. In order to perform misspeculation checking, the following coordinating state is maintained in C state memory:

(Version Numbers for Variables in D State Memory – C state of the main thread) For each variable in D state memory that is potentially read and written by parallel threads, the main thread maintains a *version number*. This version number is incremented every time the value of the variable in D state memory is modified during the committing of results produced by parallel threads. For each variable in D state memory, an associated memory location is allocated in the C state memory where the current version number for the variable is maintained.

(Mapping Table for Variables in P State Memory – C state of a parallel thread) Each parallel thread maintains a *mapping table* where each entry in the mapping table contains the following information for a variable whose value is speculatively copied from the D state memory to P state memory so that it can be used during the execution of the parallel thread computation. The mapping table is also maintained in the C state memory. As shown below, an entry in the mapping table contains five fields.

D_Addr	P_Addr	Size	Version	Write_Flag

The D_Addr and P_Addr fields provide the corresponding addresses of a variable in the D state and P state memory while Size is the size of the variable. Version is the version number of the variable when the value is copied from D state to P state memory. The Write_Flag is initialized to *false* when the value is initially copied from D state to P state memory. However, if the parallel thread modifies the value contained in P_Addr, the Write_Flag is set to *true* by the parallel thread.

Realizing CorD. When the parallel thread informs the main thread that it has completed the execution of a speculative body, the main thread consults the *mapping table* and accordingly takes the following actions. First, the main thread compares the current version numbers of variables with the version numbers of the variables in the mapping table. If some version number does not match, then the main thread concludes that misspeculation has occurred and it discards the results. If all version numbers match, then speculation is successful. Thus, the main thread commits the results by *copying* the values of variables for which the Write_flag is true from P state memory to D state memory. Note that if the Write_flag is not true, then there is no need to copy back the result as the variable's value is unchanged.

2.2.2. Optimizing Copying Operations. In the above discussion, we assumed that all data locations that may be accessed by a parallel thread have been identified and thus code can be generated to copy the values of these variables from (to) D state to (from) P state at the start (end) of parallel thread speculative body execution. Let us refer to this set as the *Copy Set*. In this section we discuss how the *Copy Set* is determined. One approach is to use compile-time analysis to conservatively overestimate the *Copy Set*. While this approach will guarantee that any variable ever needed by the parallel thread would have been allocated and appropriately initialized via copying, this may introduce excessive overhead due to wasteful copying. There are two main causes of *Copy Set* overestimation. First, even when the accesses to global and local variables can be precisely disambiguated at compile-time, it is possible that these variables may not be accessed as the instructions that access them are conditionally executed. Second, in presence of pointers, particularly when the pointers point to heap allocated data, it may not be possible to precisely disambiguate accesses to the data. In the remainder of this section we present aggressive optimizations to minimize the cost of *copying*.

Reducing Wasteful Copying. To avoid wasteful copying, we use a profile-guided approach which identifies data that is highly likely to be accessed by the parallel thread and thus potentially underestimates the *Copy Set*. The code for the parallel thread is generated such that accesses to data items are guarded by checks that determine whether or not the data item's value is available in the P state memory. If the data item's value is not available in P state memory, a *Communication Exception* mechanism is invoked that causes the parallel thread to interact with the main thread to transfer the desired value from D state memory to P state memory.

(Global/Local Variables) These variables are allocated in P state memory at the start of a parallel thread's execution. However, the values of the variables may or may not have been initialized by copying from D state memory. Therefore, a one-bit tag will be used for each variable to indicate if the variable has been initialized or not. Note that uninitialized variables, unlike initialized ones, do not have entries in the mapping table. The accesses (reads and writes) to these variables must be modified as follows. Upon a read, we check the variable's tag and if the tag is "not initialized", then the parallel thread performs actions associated with what we call *Communication Exception*.

A request is sent to the main thread for the variable's value. Upon receiving the response, which also includes the version number, the variable is allocated in P state memory, initialized using the received value, and the variable's entry in the mapping table is updated. Upon a write, the Write_flag in the *mapping table* is set and if there is no entry for the variable in the mapping table, an entry is first created.

(Heap Allocated Objects) Heap allocated objects are neither preallocated in P state memory nor are the values of their fields copied from D state memory to P state memory at the start of the parallel thread's execution. When the first time an address that corresponds to a field of a heap allocated object is assigned to a pointer variable, a communication exception occurs and the object is allocated and initialized in P state memory. When an assignment is being made via a pointer variable, the mapping table must be consulted to check if the pointer points to a copied object in P state memory or does it point to an object in D state memory. In the former case the assignment operation can proceed. However, in the latter situation actions associated with a communication exception must be performed. The parallel thread sends a request to the main thread which returns the following information: the size of the object, the values in the fields of the object, and the offset that indicates the location to which the assignment must be made in order to complete the assignment operation which caused the exception. The parallel thread uses this information to allocate an object in P state memory and set up the values of all its fields. In addition, it updates the mapping table to indicate the presence of this object in P state memory.

Optimizing Communication Exception Checks. Even for the variables which are created in P state memory at the start of a parallel thread's execution, some of the actions can be optimized. First, not all of these variables require copying in and copying out from D state memory to P state memory. Second, all the actions associated with loads and stores of these global and local variables during the execution of a parallel thread may not be required for all of the variables, i.e. some of the actions can be optimized away. As shown in Table 1, the variables are classified according to their *observed dynamic behavior* which allows the corresponding optimizations.

A *Copy In* variable is one that is observed to be *only read* by the parallel thread during profiling. Therefore its value is definitely copied in and no actions are performed at loads. However, actions are performed at stores to update the Write_flag in the mapping table so that the value can be copied out if a store is executed. A *Copy Out* variable is one that is observed to be written during its first access by the parallel thread while profiling, and thus it is not copied in but

Type of Variable	Copying Needed	Actions Needed
Copy In	Copy In = *YES*; Copy Out = *MAYBE*	Put Actions at Stores
Copy Out	Copy In = *MAYBE*; Copy Out = *YES*	Put Actions at Loads
Thread Local	Copy In = *NO*; Copy Out = *NO*	No Actions
Copy In and Out	Copy In = *YES*; Copy Out = *YES*	No Actions
Unknown	Copy In = *MAYBE*; Copy Out = *MAYBE*	All Actions

Table 1. Variable Types in Parallel Threads.

requires copying out. However, actions are needed at loads to cause a communication exception if the value is read by the parallel thread before it has been written by it. *Thread Local* variables are ones that definitely do not require either copy in or copy out, and *Copy In and Out* are variables that are always copied in and copied out. Thus, no checks are required for variables of these types. Finally, all other variables – globals/locals that are observed not to be accessed during profiling as well as all heap allocated objects accessed through pointers – are classified as *Unknown*. If these are accessed at runtime by a parallel thread, the accesses are handled via communication exceptions and thus no optimizations are possible for these variables.

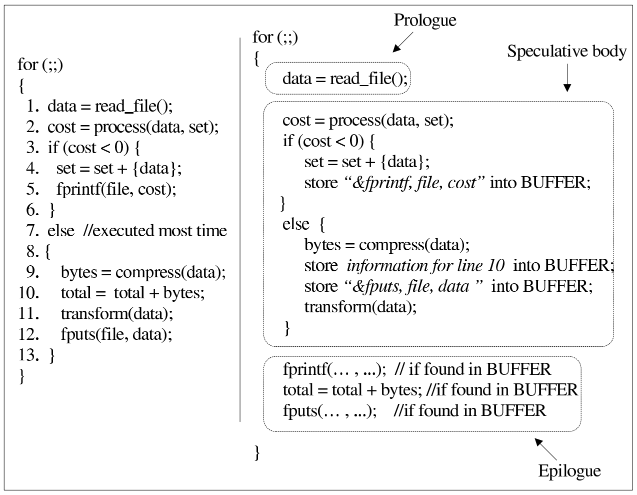
3. Speculative Parallelization

3.1. Algorithm for Partitioning a Loop Iteration

A loop iteration must be partitioned into the prologue, speculative body, and the epilogue. The algorithm for performing the partitioning first constructs the prologue, then the epilogue, and finally everything that is not included in the prologue or the epilogue is placed in the speculative body. Below we describe the construction of the prologue and the epilogue:

(Prologue) The prologue is constructed such that it contains all the input statements that read from files (e.g., fgets()). This is because such input statements should not be executed speculatively. In addition, an input statement within a loop is typically dependent *only* upon its execution in the previous iteration – this loop carried dependence is needed to preserve the order in which the inputs are read from a file. Therefore input statements for multiple consecutive loop iterations can be executed by the main thread before the speculative bodies of these iterations are assigned to parallel threads for execution. Loop index update statements (e.g. i++) are also included into the prologue, as the index variables can be considered as the input of each iteration and hence should be executed non-speculatively.

(Epilogue) The epilogue is made up of two types of statements. First, the output statements are included in the epilogue because output statements cannot be executed speculatively. If an output statement is encountered in the middle of the loop iteration or it is executed multiple times, then the code is transformed so that the results are stored in a memory buffer and the output statements that write the buffer contents to files are placed in the epilogue which is later executed non-speculatively by the main thread. Second, a statement that *may depend* upon another statement in the preceding iteration is placed in the epilogue if the probability of this dependence manifesting itself is above a threshold. Any statements that are control or data dependent upon statements already in the epilogue via an intra-iteration dependence are also placed in the epilogue.



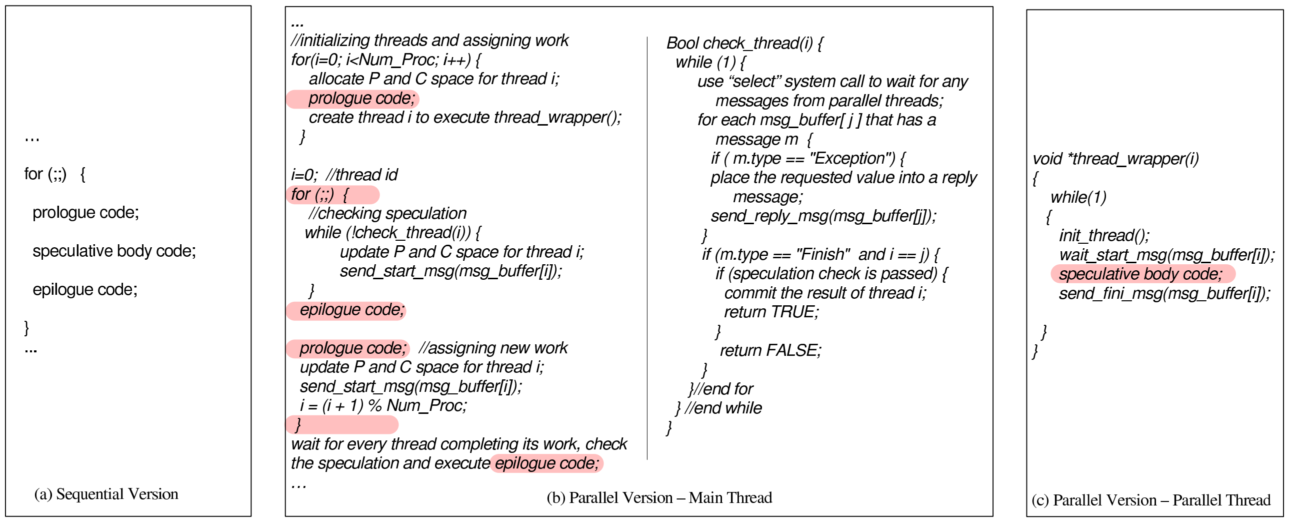
```
for (;;)
{
 1.  data = read_file();
 2.  cost = process(data, set);
 3.  if (cost < 0) {
 4.    set = set + {data};
 5.    fprintf(file, cost);
 6.  }
 7.  else  //executed most time
 8.  {
 9.    bytes = compress(data);
10.    total =  total + bytes;
11.    transform(data);
12.    fputs(file, data);
13.  }
}
```

```
for (;;)
{
   data = read_file();

   cost = process(data, set);
   if (cost < 0) {
      set = set + {data};
      store "&fprintf, file, cost" into BUFFER;
   }
   else {
      bytes = compress(data);
      store  information for line 10  into BUFFER;
      store "&fputs, file, data "  into BUFFER;
      transform(data);
   }

   fprintf(… , ...);  // if found in BUFFER
   total = total + bytes; //if found in BUFFER
   fputs(… , ...);   //if found in BUFFER
}
```

Figure 3. Partitioning a Loop into Prologue, Speculative Body, and Epilogue.

Fig. 3 illustrates the partitioning of a loop body. In the *for* loop shown on the left, the first statement is a typical input statement as it reads some data from a file and stores it into a buffer. Hence, we place it into the prologue. Then we construct the epilogue of this loop. First, all output statements (lines 5 and 12) are included. Since the profiling information can tell us that a loop dependence at line 10 is exercised very often, we also put this statement into the epilogue. If we do not do this, all speculative executions of iterations will fail because of this dependence. Thus, the epilogue of this loop has three statements, as shown by the code segment to the right in Fig. 3. Note that in this example, all three statements appear in the middle of the loop. Thus, we use a buffer to store the information of epilogue statements such as the PC of statements and values of the arguments. When the epilogue is executed by the main thread, the information stored in this buffer is referenced.

After the prologue and epilogue of a loop are identified, the rest of the code is considered as the speculative body as shown in Fig. 3. Note that line 4 may introduce loop dependence because of the accesses to variable *set*, but this



```
...
for (;;)  {
  prologue code;
  speculative body code;
  epilogue code;
}
...
```

(a) Sequential Version

```
...
//initializing threads and assigning work
for(i=0; i<Num_Proc; i++) {
    allocate P and C space for thread i;
    prologue code;
    create thread i to execute thread_wrapper();
  }

i=0;  //thread id
for (;;)  {
   //checking speculation
   while (!check_thread(i)) {
         update P and C space for thread i;
         send_start_msg(msg_buffer[i]);
    }
  epilogue code;

  prologue code;  //assigning new work
  update P and C space for thread i;
  send_start_msg(msg_buffer[i]);
  i = (i + 1) % Num_Proc;
 }
wait for every thread completing its work, check
the speculation and execute epilogue code;
...
```

```
Bool check_thread(i) {
  while (1) {
     use "select" system call to wait for any
        messages from parallel threads;
     for each msg_buffer[ j ] that has a
        message m  {
       if ( m.type == "Exception") {
       place the requested value into a reply
           message;
       send_reply_msg(msg_buffer[j]);
      }
      if (m.type == "Finish"  and i == j) {
        if (speculation check is passed) {
          commit the result of thread i;
          return TRUE;
        }
         return FALSE;
      }
    }//end for
  } //end while
}
```

(b) Parallel Version – Main Thread

```
void *thread_wrapper(i)
{
   while(1)
   {
     init_thread();
     wait_start_msg(msg_buffer[i]);
     speculative body code;
     send_fini_msg(msg_buffer[i]);

   }
}
```

(c) Parallel Version – Parallel Thread

Figure 4. Code Transformation.

dependence seldom manifests itself. So we actually speculate on this variable. It is worth noting that placing line 4 into the epilogue does not help the parallelism of the loop, because the variable *set* is used by function *process* in every iteration. If this variable is changed, whether by parallel threads or the main thread, all subsequent iterations being executed will have to redo their work.

3.2. Parallelizing Transformation

Next we show the form of the main thread and the parallel thread created by our speculative parallelization transformation. Before we present the detailed transformed code, let us see how the main thread and parallel threads interact. The main thread and a parallel thread need to communicate with each other to appropriately respond to certain events. This communication is achieved via messages. Four types of messages exist. When the main thread assigns an iteration to a parallel thread, it sends a Start message to indicate to the parallel thread that it should start execution. When a parallel thread finishes its assigned work, it sends a Finish message to the main thread. When a parallel thread tries to use a variable that does not exist in the P space, a communication exception occurs which causes the parallel thread to send an Exception message to the main thread. The main thread services this exception by sending a Reply message.

The main thread allocates a *message buffer* for each parallel thread it creates. This message buffer is used to pass messages back and forth between the main thread and the parallel thread. When a parallel thread is free, it waits for a Start message to be deposited in its message buffer. After sending an Exception message, a parallel thread waits for the main thread to deposit a Reply message in its message buffer. After sending Start messages to the parallel threads, the main thread waits for a message to be deposited in any message buffer by its parallel thread (i.e., for a Finish or Exception message). Whenever the main thread encounters an Exception message in a buffer, it processes the message and responds to it with a Reply message. If a message present in the message buffer of some parallel thread is a Finish message, then the main thread may or may not process this message right away. This is because the results of the parallel threads must be *committed in order*. If the Finish message is from a parallel thread that is next in line to have its results committed, then the Finish message is processed by the main thread; otherwise the processing of this message is postponed until a later time. When the main thread processes a Finish message, it first checks for misspeculation. If misspeculation has not occurred, the results are committed and new work is assigned to the parallel thread, and a Start message is sent to it. However, if misspeculation is detected, the main thread prepares the parallel thread for reexecution of the assigned work and sends a Start message to the parallel thread. The above interactions continue as long as the parallel threads continue to speculatively execute iterations.

Another important issue is the number of parallel threads that should be created. One approach is to dedicate one core to the execution of the main thread and create one parallel thread for each additional core available. However, we observe that while the parallel threads are executing, the main thread only needs to execute when a parallel thread sends a message to it. In addition, the parallel thread sending the message must wait for the main thread to respond before it can continue execution. Thus, we do not need to dedicate a core for the main thread. Instead we can create as many parallel threads as there are cores available and the main thread can execute on the same core as the parallel thread with which it is currently interacting through messages. This strategy can be implemented in POSIX as follows. The main thread executes the `select` call which allows it to wait on multiple buffers corresponding to the multiple parallel threads. The call causes the main thread to be descheduled and later woken up when a message is written into any of the message buffers being monitored. Unlike the main thread, a parallel thread simply needs to wait for the main thread. Therefore, a parallel thread can execute the `read` call which allows the parallel thread to monitor a single message buffer which is its own message buffer. Upon executing a `read`, the parallel thread is descheduled until the main thread performs a write to the monitored buffer.

Having discussed how the main thread and parallel threads interact and work together to execute speculatively parallelized code, now we show the corresponding code transformation. Fig. 4(a) shows a sequential version of the code. After transformation, the structure of the resulting code for the main thread and a parallel thread are shown in Fig. 4(b) and (c) respectively. As we can see, the code executed by the main thread first creates parallel threads and allocates P space and C space. Next, the main thread enters a loop where it first calls `check_thread` which takes the following actions. It processes Exception requests from all parallel threads until finally a Finish message is received from the parallel thread that executed the earliest speculative iteration currently assigned to the parallel threads (this is thread `i` in the code). Upon receiving this message, a check is made to detect *misspeculation*. If speculation is successful, the results are committed and the `check_thread` routine returns TRUE; otherwise it returns FALSE and as a result the main thread prepares the parallel thread for reexecuting the assigned iteration. As we can see, once the results have been committed, the *epilogue code* is executed. Next, the main thread executes the *prologue code* for the next available iteration and prepares the idle parallel thread to execute this iteration. Finally, the value of `i` is updated to identify the next parallel thread whose results will be committed by the main thread.

```
if (b is not initialized) {
    msg.p_addr = &b;
    msg.len      =  sizeof(b);
    msg.status   = "Exception";
    send(msg_buffer, msg);
    b = read(msg_buffer);
    ver = read(msg_buffer);
    update_mapping_table(&b,
                    sizeof(b), ver);
}

a = b + 1; // intermediate code

r = search_mapping_table(&a);
if (r == NULL) {
    r = update_mapping_table(&a);
}
r.write_flag = 1;
```

(a) Handling Loads and Stores

```
p = expr;  // p is a pointer

r = search_mapping_table(p);
if (r == NULL)
{
    msg.d_addr = p;
    msg.len      =  PTR_LEN;
    msg.status   = "Exception";
    send(msg_buffer, msg);
    len = read(msg_buffer);
    p = malloc(len);
    *p = read(msg_buffer);
    version = read(msg_buffer);
    update_mapping_table(p, len, ver);
    offset = read(msg_buffer);
    p += offset;
}
```

(b) Handling Pointers

Figure 5. Parallel Thread: Detecting and Handling Communication Exceptions.

Now let us consider further details of the *speculation body code* in the parallel thread. Recall that the code must be modified to guard data accesses appropriately so that communication exceptions are appropriately generated and the mapping table is maintained. Although the code that is generated will depend upon the types of variables shown in Table 1, in Fig. 5, we show the generated code for the situation in which no optimization is possible, i.e. the variable type is *Unknown*. For other variable types, an appropriate subset of this code is generated. In Fig. 5(a) we consider the execution of statement `a=b+1`. The code preceding the statement handles the loading of `b` and the code following the statement handles the storing of the value of `a`. Fig. 5(b) considers the case in which an assignment to pointer variable `p` is made. The code following the assignment determines if the address assigned corresponds to a heap object that must be copied from D space into P space.

3.3. Enhancements

(Reducing Thread Idling) The performance of the parallel version may still be hindered by two factors. The first one is *thread idling*. If a parallel thread that is assigned work earlier, finishes its work before some later threads get their work from the main thread, it has to wait for the main thread to check its result. However, it may take a long time for the main thread to finish assigning the later iterations to other threads. So during this period, this parallel thread cannot do any other work but simply idle. This will cause substantial performance loss. To avoid thread idling, we can increase the workload of each parallel thread by assigning 2 or more iterations at one time using loop unrolling. In this way, we ensure every thread stays busy while the main thread is assigning the later iterations to other threads.

(Delayed Coping) Another factor that affects the performance is the *misspeculation rate*. Since our approach speculates on rarely-exercised loop-carried dependencies, a misspeculation may cause the result of some later threads to be discarded. Therefore, an earlier thread's misspeculation can lead to significant waste when more threads are used. To reduce the performance impact of misspeculations, we delay copying the variables on which we speculate. Specifically, the values of these variables will not be copied from D space until the variables are accessed for the first time. The idea of *delayed copying* is to increase the chance of obtaining the correct version of these variables as some earlier thread may commit its values.

4. Experiments

To speculatively parallelize loops, we first profile the loop code to gather information such as the dependence graph and dynamic access patterns of variables. In our implementation we use the Pin [16] instrumentation framework to enable profiling. Since the output of Pin is in the form of instruction

Program	LOC	Loop Exec. Time	Prof. Input	Exp. Input	Prologue	Epilogue	Vars. in Body			
							I	O	L	IO
197.parser	9.7K	100%	1K file	36K file	fgets	printf	49	6	12	2
181.mcf	1.5K	31%	test/inp.in	train/inp.in	i++	total++	5	0	5	3
130.li	7.8K	100%	6 scripts	72 scripts	i++	printf var++	30	0	3	6
256.bzip2	2.9K	100%	200K file	7.5M file	fgetc	fputc	12	8	11	1
255.vortex	49.3K	80%	test/input	train/input	++i	fprintf	76	5	4	6
CRC32	0.2K	100%	one 4M-file	80 4M-files	−−argc	printf	1	0	2	1

I-*Copy In* O-*Copy Out* L-*Thread Local* IO-*Copy In and Out*

Table 2. Characteristics of Benchmarks.

or memory addresses, we use the output of *objdump*, a utility that can display the information from object files, to map these addresses to the names of variables and statements in the original programs. We make the stack variables *static* so that they get stored in the data or bss segment and hence can be recognized by objdump. Based upon the gathered profile data, the compiler performs code transformation. We choose LLVM [15] as our compiler infrastructure as it allows us to perform static analysis and customized transformation. All code transformations are performed at the level of *bc* code, the intermediate representation of LLVM. After the transformation, we use the *native* option of LLVM to generate the x86 binary code which can be executed on the real machine. All our experiments were conducted under Fedora 4 OS running on a dual quad-core 3.0 GHz Xeon machine with 16GB memory.

4.1. Case Studies

Table 2 describes the programs used to evaluate our parallelization approach. Among the 6 programs used, 5 are from SPEC [10] and 1 from MiBench [8]. In the table, the first two columns show the name and the number of source code lines of each program. The third column shows the execution time of the loops that we parallelized as a percentage of total program execution time – as we can see, the loops represent the entire execution time in all but one program. The profiling input is shown by column *Prof. Input* and the column *Exp. Input* gives the input used in the experimental evaluation. The next two columns show the contents of the prologue and epilogue of the parallelized loop. We also use profiling to identify different communication types of each variable used in the speculative body. The last four columns show the distribution of variables across these categories (recall these variables do not include heap data). *All these programs are the ones where without speculation, parallelization is not possible.*

197.parser is a program that can parse a given sentence based on the link grammar. In its *batch* mode, a *while* loop is used to parse a given file line by line, one sentence per line. After the loop partition, the speculative body parses a sentence. If a sentence conforms to a certain pattern, a global variable is changed indicating that the parsing process of the following lines must be different. Also, if a sentence is not consistent with the grammar, a variable *errors* is incremented by 1. According to the profiling result, these two cases do not appear very often and hence we can speculate on these two variables.

181.mcf is a program for solving the single-depot vehicle scheduling problem. A hot loop is identified in function *price_out_impl* and is parallelized. In its speculative body, a variable *red_cost* is computed over a list. If this variable is less than 0, a new arc will be inserted into the list, and two counters, *dep and new_arc*, will be updated. Since most of the time this branch will not be taken, we can speculate on the tail pointer of this list and these two counters.

130.li is an interpreter of the extended Lisp language which supports object-oriented programming. It uses a *for* loop to process a set of Lisp scripts specified on the command line. Since these scripts can be interpreted separately, we exploit the parallelism of this loop. The profiling result actually shows that besides I/O stream dependencies, there exist loop dependencies on 12 variables. Among these variables, 6 are used to count the number of certain types of memory allocations. These dependences are exercised between every two consecutive iterations, so we put the corresponding statements into the epilogue so that they are executed by the main thread non-speculatively. For the remaining 6 variables, 5 are used to store information such as context, stack pointer, or environment vectors when each script is processed. Although these dependencies occur in every iteration, they are all silent, meaning that at the beginning and the end of each iteration, the values of each variable are the same. In our experiments, we specify them as *copy in and out* type. This is because we perform a value-based speculation check which can effectively prevent these silent dependencies from serializing the loop execution. The last one is the variable *xldebug*, which is updated when there is an error in a script. Since errors do not appear very often, we speculate on this variable. Thus, we can see that 6 *copy in and out* variables are found in the speculative body.

256.bzip2 is a data compressor which uses Huffman encoding and Burrows-Wheeler transformation. During the compression process, a *while* loop is used in function *compressStream* to compress a file. Within this loop, a fixed-size

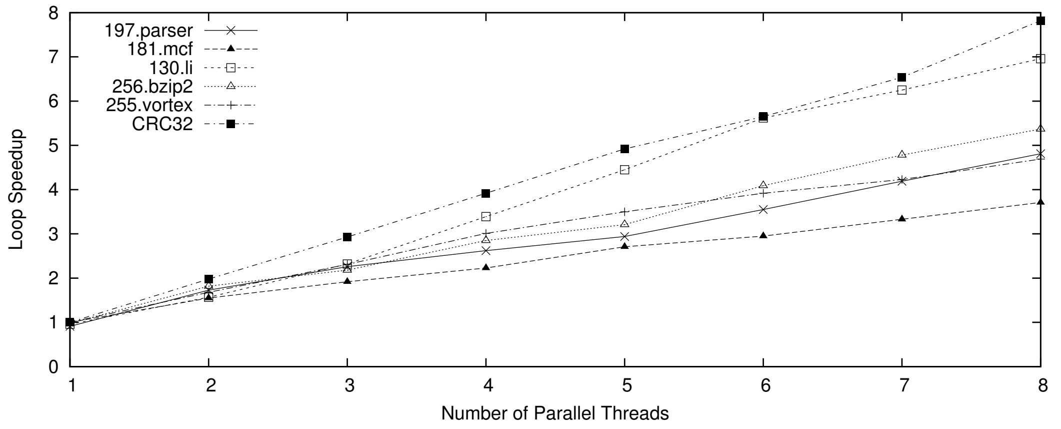


Figure 6. Execution Speedups.

block is first read from a file stream, then compressed and written to an output file. Thus, this loop can be executed in parallel as the blocks can be independently compressed. The speculative body performs compression. If an exception occurs during the compression, the program will jump to the corresponding handler. In our parallel code, we do speculation on those checks. If any exception occurs, the main thread stops all the worker threads executing the subsequent iterations and then processes the exception.

255.vortex is a database transaction benchmark. This program's execution performs a certain number of inserts, deletes and lookups on a pre-configured database. In our experiments, the *while* loop in function *BMT_LookUpParts* can be identified and parallelized. In the speculative body, an object is selected randomly and searched from the database. During this process, a variable *Status* will be updated if any error is encountered. Also, when certain conditions become true, several assertion variables will be updated accordingly. However, since all these situations rarely happen, we speculate on these variables. Note that random selection of objects is implemented through the function *Random*, which causes a loop-carried dependence on variable *jran*. This dependence actually can be ignored because this function is *commutative* [2], [14]. In our framework, the compiler does not generate misspeculation checks for variables used in commutative functions.

CRC32 is a program used to compute the 32-bit CRC of a file. It accepts multiple file names as the input and uses a *for* loop to calculate the CRC of those files one by one. Since the calculation for each file is independent, this loop can be parallelized. The prologue of this loop is a typical loop index statement and the epilogue is a *printf* call. The rest of the statements are the speculative body. In this body, a variable *error* will be updated if any problems are encountered during the CRC computation. However, the value of this variable changes infrequently and thus we speculate on it.

4.2. Execution Speedups

To evaluate the performance of our model for every benchmark, we first measured the baseline which is the sequential execution time of the loop that was parallelized. Then we measured the time of executing this loop in our model with different numbers of parallel threads. Fig. 6 shows our best result. Each execution time is obtained by taking the average of the execution times of 10 runs. In all cases, the copy communications between the main thread and parallel threads are optimized.

From Fig. 6, we can see that when the number of parallel threads increases, the loop speedup for all benchmarks goes up linearly. The highest speedup achieved ranges from 3.7 to 7.8 across the benchmarks when 8 parallel threads are used. We also notice that the performance of our model with one parallel thread is slightly worse than with the sequential version for some benchmarks. That is due to the copying, tracking and checking overhead. In fact, if only one core is available, we can force the main thread to perform all computations rather than spawning parallel threads.

4.2.1. Benefits of Enhancements. The data shown in Fig. 6 was obtained after applying all performance enhancements. We found that idling threads and misspeculation can significantly affect the speedups for all benchmarks except for CRC32 when more than 5 threads are used. In particular, when using the basic transformation, we noticed that idling threads caused performance to remain unchanged for 197.parser, 130.li, 256.bzip2, and 255.vortex when the thread number is above 5 or 6, and using additional threads did not increase the speedup further. Table 3 shows the speedups of these benchmarks with and without reducing thread idling. Clearly, increasing the workload of parallel threads takes effect when the number of parallel threads is more than 5 (197.parser, 130.li and 256.bzip2), or 6 (255.vortex). In the case of CRC32, however, we are still able to obtain more speedups with more threads when using the basic transformation because each thread performs substantial computation on a 40M file.

Threads		197.parser	130.li	256.bzip2	255.vortex
4	w/o	2.63	3.27	2.72	3.02
	w/	2.62	3.39	2.85	3.01
5	w/o	2.70	3.92	2.80	3.43
	w/	2.94	4.45	3.21	3.50
6	w/o	2.77	4.59	2.99	3.72
	w/	3.55	5.62	4.09	3.90
7	w/o	2.87	4.61	2.98	3.92
	w/	4.19	6.25	4.78	4.23
8	w/o	2.90	4.63	2.98	3.95
	w/	4.81	6.96	5.37	4.69

Table 3. Effect of Reducing Thread Idling on Speedups.

For benchmark 181.mcf, we found that the misspeculation rate, which increases from 0.7% to 17.5% as the number of threads increases from 2 to 8, limited the speedup. For other benchmarks, however, the misspeculation rate stays below 2% and does not vary much and hence it does not impact performance. Table 4 shows the speedup of 187.mcf with and without delayed copying. As we can see, this technique can provide moderate improvement in the performance. This is because with this technique, the misspeculation rate of this benchmark is not more than 10%.

Threads	w/o Delayed Copying	w/ Delayed Copying
2	1.45	1.55
3	1.79	1.92
4	2.03	2.23
5	2.41	2.71
6	2.65	2.95
7	2.99	3.33
8	2.72	3.71

Table 4. Effect of Delayed Copying on Speedups for 181.mcf.

4.2.2. Benefits of Copy Optimization. As shown in Table 2, we use profiling information to identify the communication type of each variable so that the copy in and out overhead can be minimized. To evaluate the effect of our optimization, we compare the performance of the optimized program with two other versions of the program: (Copy-All) that uses copy-in of all variables that will be potentially used by parallel threads; and (Copy On-the-fly) in which no variables are initially copied, i.e. each variable is copied on-the-fly using a communication exception upon the first access to it. While Copy-All may perform excessive copying and spends more time maintaining the mapping table, Copy On-the-fly will not perform unnecessary copying but will execute extra checks (e.g., even for variables that could have been made thread local). Copy operations of heap data are performed on the fly for all schemes.

Table 5 shows the loop speedups for each benchmark with different copy schemes. According to the results shown, our optimized copy scheme outperforms the other two schemes in all cases except for CRC32. In the case of CRC32, the computation only uses 4 variables, so all copy schemes perform about the same. We also observed that the Copy On-the-fly scheme for 256.bzip2 and 187.mcf is almost as good as the optimized one. For 256.bzip2, the reason is that most memory accesses in this benchmark are heap references, and all schemes copy the heap data on-the-fly. For 187.mcf, Copy On-the-fly reduces misspeculation rate which is the primary limiting factor for its performance.

Program	Copy All	Copy On-the-fly	Opt.
197.parser	2.28	2.38	2.63
130.li	2.68	2.67	3.09
256.bzip2	2.35	2.62	2.72
181.mcf	1.90	2.21	2.23
255.vortex	2.50	2.33	3.01
CRC 32	3.92	3.86	3.91

Table 5. Loop Speedup for Different Copy Schemes with 4 Parallel Threads.

We observe that the performance of the Copy-All and the Copy On-the-fly schemes differs for different benchmarks. In some cases (197.parser, 256.bzip2) the Copy On-the-fly scheme performs much better than the Copy-All scheme. This is because the Copy On-the-fly scheme does not perform any copy operations for thread local variables. Using more thread local variables will only affect the Copy-All scheme, not the Copy On-the-fly scheme.

Note that the data shown in this figure is based on executions using 4 parallel threads. We also conducted the experiments with different numbers of parallel threads. The results observed are similar. This is because the total number of iterations that need to be executed are the same, and for each copy scheme, the same amount of copy operations need to be performed over the entire execution.

4.3. Overheads

4.3.1. Time Overhead. Our software speculative parallelization technique involves overhead due to instructions introduced during parallelization. We measured this execution time overhead in terms of the fraction of total instructions executed on each core. The results are based upon an experiment in which we use 8 parallel threads, and we breakdown the overhead into five categories as shown in Table 6. The second column *Static Copy* is the fraction of the total number of instructions used for performing copy-in and copy-out operations by the main thread. This overhead ranges from 0.02% to 5.28% depending on how many variables need to be copied. The third column *Dynamic Copy* gives the fraction of instructions for on-the-fly copying. The *Exception Check* column shows the fraction of instructions used by parallel threads to check if a variable has been copied into the local space. According to the results, these two numbers are very low for the benchmarks we used. Another category of overhead comes from the *Misspeculation Checking*. This uses 1%-2% instructions for all benchmarks except for CRC32 which does not have many variables to copy. Besides the above four categories, there are other instructions executed for *Setup* operations (e.g., thread initialization, mapping table allocation and deallocation etc.). The last column shows the result. In total, no more than 7% of total instructions are used for execution model on each core.

Program	Static Copy	Dynamic Copy	Exception Check	Misspec. Check	Setup
197.parser	3.51%	0.33%	0.02%	1.76%	0.62%
130.li	0.08%	0	0	1.08%	0.07%
256.bzip2	1.32%	0.25%	0.06%	1.03%	0.48%
181.mcf	1.97%	0.13%	0.08%	2.81%	2.15%
255.vortex	5.28%	0.04%	0.01%	1.25%	0.39%
CRC32	0.02%	0	0	0.01%	0.32%

Table 6. Overhead Breakdown on Each Core.

4.3.2. Space Overhead. Since we partition the memory into three states during the execution, and each parallel thread has its own C and P state memory, extra space certainly needs to be used in our execution model. So we measured the space overhead of the executions of parallelized loops. The space overhead is shown in Table 7. The space used by the sequential version serves as the baseline.

Program	1 thread	2 threads	4 threads	8 threads
197.parser	1.01	1.22	1.46	2.13
130.li	1.14	2.56	1.95	2.95
256.bzip2	1.32	2.13	3.86	5.81
181.mcf	1.05	1.13	1.36	2.29
255.vortex	1.14	1.36	1.57	1.95
CRC32	1.04	1.38	2.01	3.28

Table 7. Memory Space Overhead.

As we can see, the overhead for most benchmarks is around 2x-3x when 8 threads are used. Given the speedup achieved for these benchmarks, we can see that the memory overhead is acceptable. For 256.bzip2, a large chunk of heap memory allocated in D space is used during the compression. In our execution model, each parallel thread will make a copy of this memory space to execute the speculative body. Therefore, as more parallel threads are used, more memory is consumed.

Program	Sequential Version	Parallel Version
197.parser	234K	239K
130.li	179K	183K
256.bzip2	53K	57K
181.mcf	22K	24K
255.vortex	1336K	1370K
CRC32	8K	10K

Table 8. Size of Binary Code.

Besides the dynamic space consumption, we also examined the increase in the static size of the binary. As shown in Table 8, the increase varied from 2K to 5K for most programs, a very small fraction of the binary size.

5. Related Work

Ding et al. [6] proposed a *process based* runtime model that enables speculative parallel execution of Potentially Parallel Regions (PPRs) on multiple cores. Due to use of processes, significant amount of memory pages need to be copied when speculation succeeds. The parallelism is not fully exploited in this scheme because the work is assigned to the cores round by round and the next round cannot start until all work in the previous round is finished successfully. If speculation with respect to a process fails, the work by following processes in the same round is completely discarded. None of the above drawbacks are present in our approach. Finally, brute force methods are used to identify the PPRs in [6] while in our work we divide all statements in a loop into three partitions according to their dynamic execution pattern. Kulkarni et al. [13], [14] proposed a runtime system to exploit the data parallelism in applications with irregular parallelism. Parallelization requires speculation with respect to data dependences. The programmer uses two special constructs to identify the data parallelism opportunities. When speculation fails, user supplied code is executed to perform rollback. In contrast, our work does not require help from the user, nor does it require any rollbacks. Finally, an important aspect of the above work is its use of commutativity property – a function is considered to be commutative if its calls can be executed in different order without changing the correctness of the program. Commutative functions either do not introduce any loop-carried dependencies or these dependencies can be safely ignored. In our work commutativity can be exploited by eliminating misspeculation checks – recall that in 255.vortex the calls to a random number generator were commutative.

One commonly-used approach for parallelization of loops is software pipelining. This technique partitions a loop into multiple pipeline stages where each stage is executed on a different processor. Decoupled software pipelining (DSWP) [18], [19], [22] is a technique that targets multicores. The proposed DSWP techniques require two kinds of hardware support that is not commonly supported by current processors. First, hardware support is used to achieve efficient message passing between different cores. Second, hardware support is versioned memory which is used to support speculative DSWP parallelization. Since DSWP requires the flow of data among the cores to be acyclic, in general, it is difficult to balance the workloads across the cores. Raman et al. [19] address this issue by parallelizing the workload of overloaded stages using DO-ALL techniques. This technique achieves better scalability than DSWP but it does not support speculative parallelization which limits its applicability. Other recent works on software pipelining target stream and graphic processors [3], [7], [11], [12], [21].

The alternative approach to exploiting loop parallelism is DO-ALL technique [6], [14], [13], [5], [9], [23], [20], [1],

[17], [25] where each iteration of a loop is executed on one processor. Among these works, a large number of them focus on thread level speculation (TLS) which essentially is a hardware-based technique for extracting parallelism from sequential codes [5], [9], [23], [20], [1], [17], [25]. In TLS, speculative threads are spawned to venture into unsafe program sections. The memory state of the speculative thread is buffered in the cache, to help create thread isolation. Hardware support is required to check for cross thread dependence violations; upon detection of these violations, the speculative thread is squashed and restarted on the fly. Compared to TLS, our work does not require any hardware support and can be done purely in software.

Vijaykumar et al. [24] also presented some compiler techniques to exploit parallelism of sequential programs. A set of heuristics operate on the control flow graph and the data dependence graph so that the code can be divided into tasks. These tasks are speculatively executed in parallel and the hardware is responsible for detecting misspeculation and performing recovery. However, this work focuses specifically on Multiscalar processors. Instead of concentrating on extracting coarse-grained parallelism, Chu et al. [4] recently proposed exploiting fine-grained parallelism on multicores. Memory operations are profiled to collect memory access information and this information is used to partition memory operations to minimize cache misses.

6. Conclusion

We presented a novel Copy or Discard (CorD) execution model to efficiently support software speculation on multicore processors. The state of speculative parallel threads is maintained separately from the non-speculative computation state. The computation results from parallel threads are committed if the speculation succeeds; otherwise, they are simply discarded. A profile-guided parallelization algorithm and optimizations are proposed to reduce the communication overhead between parallel threads and the main thread. Our experiments show that our approach achieves speedups ranging from 3.7 to 7.8 on a server with two Intel Xeon quad-core processors.

Acknowledgment

This work is supported by NSF grants CNS-0810906, CNS-0751961, CCF-0753470, and CNS-0751949 to the University of California, Riverside.

References

[1] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA*, pages 99–108, 2002.
[2] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multicore. In *MICRO*, pages 69–84, 2007.
[3] I. Buck. *Stream computing on graphics hardware*. PhD thesis, Stanford, CA, USA, 2005.
[4] M. Chu, R. Ravindran, and S. Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *MICRO*, pages 369–380, 2007.
[5] M. H. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA*, pages 13–24, 2000.
[6] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.
[7] K. Fan, H. Park, M. Kudlur, and S. A. Mahlke. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *CGO*, pages 124–133, 2008.
[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
[9] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, pages 58–69, 1998.
[10] http://www.spec.org.
[11] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.
[12] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI*, 2008.
[13] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS*, pages 233–243, 2008.
[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.
[15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, page 75, 2004.
[16] 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*, pages 190–200, 2005.
[17] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS*, pages 365–372, 1999.
[18] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, pages 105–118, 2005.
[19] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO*, pages 114–123, 2008.
[20] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, pages 1–12, 2000.
[21] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO*, pages 356–369, 2007.
[22] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT*, pages 49–59, 2007.
[23] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. S. Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.
[24] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *MICRO*, pages 81–92, 1998.
[25] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *MICRO*, pages 85–96, 2002.