Int J Parallel Prog (2009) 37:508–535
DOI 10.1007/s10766-009-0111-z

Speculative Parallelization of Sequential Loops on Multicores

Chen Tian · Min Feng · Vijay Nagarajan · Rajiv Gupta

Received: 12 January 2009 / Accepted: 31 May 2009 / Published online: 19 June 2009

Abstract The advent of multicores presents a promising opportunity for speeding up the execution of sequential programs through their parallelization. In this paper we present a novel solution for efficiently supporting software-based speculative parallelization of sequential loops on multicore processors. The execution model we employ is based upon *state separation*, an approach for separately maintaining the speculative state of parallel threads and non-speculative state of the computation. If speculation is successful, the results produced by parallel threads in speculative state are committed by copying them into the computation's non-speculative state. If misspeculation is detected, no costly state recovery mechanisms are needed as the speculative state can be simply discarded. Techniques are proposed to reduce the cost of data copying between non-speculative and speculative state and efficiently carrying out misspeculation detection. We apply the above approach to speculative parallelization of loops in several sequential programs which results in significant speedups on a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors.

Keywords Multicores · Speculative parallelization · Profile-guided parallelization · State separation

C. Tian (✉) · M. Feng · V. Nagarajan · R. Gupta
Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA, USA
e-mail: tianc@cs.ucr.edu

M. Feng
e-mail: mfeng@cs.ucr.edu

V. Nagarajan
e-mail: vijay@cs.ucr.edu

R. Gupta
e-mail: gupta@cs.ucr.edu

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 frequency with which at which speculation is successful.

In this paper we present a novel approach for effectively addressing the aforementioned issues. This approach employs an execution model for speculative execution in which 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. A key feature of this model is state separation according to which 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 handle 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*. 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.

We 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 speculative parallelization of loops in several sequential applications achieves significant speedups (up to a factor of 7.8) on a Dell PowerEdge 1900 server with eight cores in form of two 3.00 GHz Intel Xeon quad-core processors and 16 GB of RAM.

The remainder of the paper is organized as follows. In Sect. 2 our approach to speculative execution is described. In Sect. 3 we present the details of how loops in a sequential application are transformed for speculative parallel execution. The results of our experiments are presented in Sect. 4. The related work is discussed in Sect. 5. Section 6 summarizes our conclusions.

2 Speculative Parallel Execution Model

In this section we describe the key aspects of our approach to speculative execution and identify the compiler and runtime support needed to realize the model. Consider a pair of subcomputations C and C' in a sequential program P such that the execution of C precedes the execution of C', i.e. $C \rightarrow C'$ during sequential execution (e.g., C and C' may represent consecutive iterations of a loop). Thus, the results computed during the execution of C are available during the execution of C'. The goal of *speculative execution* is to relax the strict ordering imposed on the execution of C and C' by speculatively executing C' while C is still executing. During the speculative execution of C', if a data value is read prematurely (i.e., it is read before it has been computed by C), then *misspeculation* occurs and thus the results computed during speculative execution of C' must be discarded and C' must be executed again. On the other hand, if misspeculation does not occur, the execution time of the program is potentially reduced due to parallel execution of C and C'. Of course, speculative execution is only beneficial if the misspeculation occurs *infrequently*. Opportunities for speculative execution arise because the dependences from C to C' may arise from infrequently executed code or even if they do arise, they may be deemed harmless (e.g., dependences may arise due to silent stores). The above approach naturally extends to multiple levels of speculation. Given a series of dependent computations $C_1 \rightarrow C_2 \rightarrow \cdots C_n$, while C_1 executes non-speculatively, we can speculatively execute C_2 through C_n in parallel with C_1 on additional cores.

We propose an execution model, that supports speculative execution as illustrated above, with three key characteristics that make its realization amenable via compiler and runtime support. In the remainder of this section we describe these characteristics.

2.1 State Separation

The first key characteristic of our execution model is *state separation* according to which the non-speculative state of the program is maintained separately from the speculative state of the computation. 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 parts of the speculative state of the computation. State separation guarantees thread isolation, i.e. the execution of each thread, main or parallel, is isolated from execution of all other threads. Thus, if execution of one thread results in misspeculation, it does not necessarily force the reexecution of other threads.

To implement state separation, the compiler and runtime must implement a speculative computation such that it is allocated separate memory and all manipulations of data performed by the speculative computation is performed in this separate memory. Therefore, shared data that is manipulated by a speculative computation is *copied from* non-speculative state to speculative state and then manipulated by the speculative computation in the speculative state. Finally, if misspeculation does not occur, the updated speculative state is *copied to* the non-speculative state; else, the speculative state is

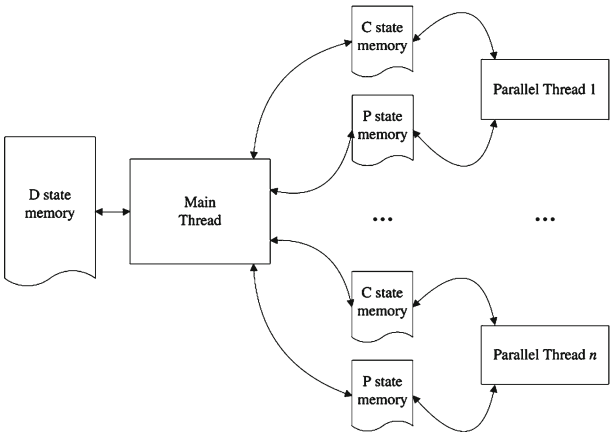


Fig. 1 Maintaining memory state

simply *discarded*. We refer to this approach for handling the results of a speculative computation as *Copy or Discard* (*CorD*). This approach extends to multiple levels of speculation. Given a series of dependent computations $C_1 \rightarrow C_2 \rightarrow \cdots C_n$ such that, while C_1 executes non-speculatively, if we speculatively execute C_2 through C_n in parallel with C_1, then the results of C_2 through C_n must be copied to the non-speculative state *in-order*. This will ensure that if any dependences arise between C_i and C_j (for all $i < j$), they are correctly enforced.

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. 1).

(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. 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 Misspeculation: Detection & Recovery

The second key characteristic of our execution model is the manner in which misspeculation can occur. Given dependent computations $C \rightarrow C'$, when we start speculative execution of C' in parallel with C, we are essentially speculating that no dependences exist from C to C'. In other words, we are speculating that the values that are copied from non-speculative state to speculative state of C' will not be modified by C. Therefore misspeculation detection must perform the following task. At the end of the execution of C', we must check to see if C modified any of the values that were speculatively read by C'. If this is indeed the case, misspeculation occurs and recovery is performed by reexecuting C'. Otherwise speculation is successful and the results computed by C' are copied back to non-speculative state. 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.

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.

2.3 Optimizing Copying Operations

Since *state separation* presents a clear and simple model for interaction between the non-speculative and speculative computation state, it is amenable for implementation through compiler and runtime support. However, this simplicity comes at the cost of *copying overhead*. Therefore, to achieve high efficiency, the combination of compiler and runtime support must be developed. The compiler can play an important role in minimizing copying overhead. Through compile time analysis we will identify the subset of non-speculative state that must be copied to the speculative state. In particular, if the compiler identifies data items that are *not referenced* by the speculative computation, then they need not be copied. Moreover, shared data that is *only read* by the speculative computation, need not be copied as it can be directly read from the non-speculative state.

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. The main causes of *Copy Set* overestimation is that 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 may not be executed.

(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. Specifically, 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 in P state memory is 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.

(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 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*

Table 1 Variable types in parallel threads

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

are variables that are always copied in and copied out. Thus, no checks are required for variables of these types. Finally, all other variables that are observed not to be accessed during profiling 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 of Loops

As we have seen, 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. In this section we show how this approach can be applied to speculative parallelization of loops.

Figure 2 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. 2) 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.

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

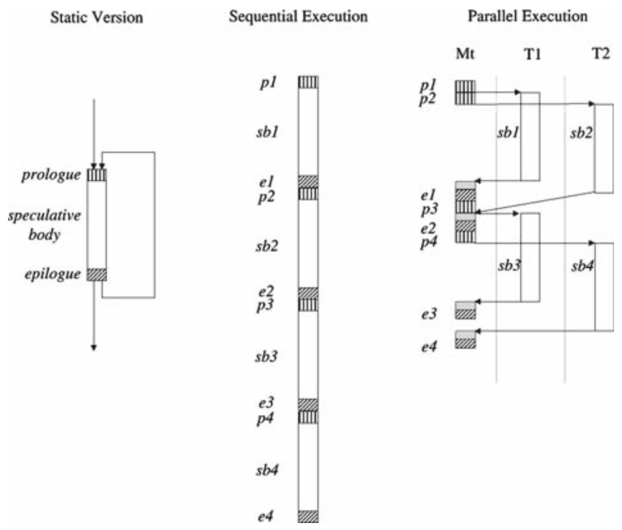


Fig. 2 Thread execution model

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.

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 by reexecuting the loop iteration involved.

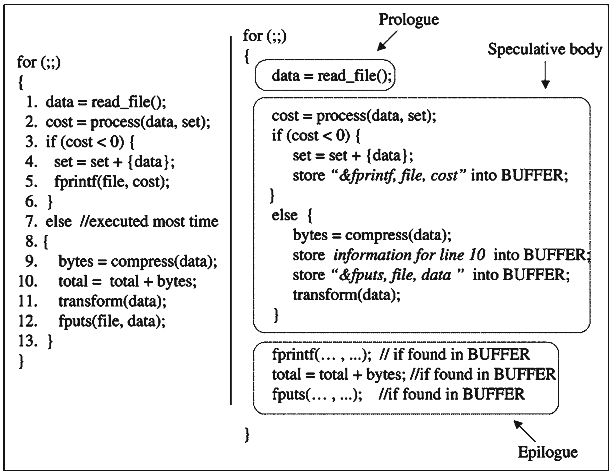
3.1 Algorithm for Partitioning a Loop Iteration

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.

Figure 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



```
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
}
```

Fig. 3 Partitioning a loop into prologue, speculative body, and epilogue

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 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 Transforming Partitioned Loop

Next we show the detailed form of the main thread and the parallel thread created by our speculative parallelization transformation. We first discuss how the threads interact with each other and then we present the details of the work carried out by each of the threads.

(Thread Interaction) 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.

(Main Thread) The code corresponding to the main thread is shown in Fig. 4a. In addition to executing the prologue and epilogue code, the main thread performs the following actions. It calls `init_version_table()` [see Fig. 4b] to initialize the version table that it must maintain. Next it creates parallel threads one at a time and initializes them by calling `init_thread` [see Fig. 4c]. During the first call to the initialization routine corresponding to a parallel thread, the main thread allocates the C space for the mapping table and initializes the mapping table. In subsequent calls to the initialization routine for the same thread, the mapping table is only initialized since it has already been allocated. Copy operations are also performed by the main thread for the variables that are marked as `Copy In` and `Copy In and Out`. Note that the P space into which values are copied is allocated when a parallel thread is created. For `unknown` variables, we do not setup any record in the mapping table. Instead, we just initialize these variables' one-bit tags to false. The main thread also gets a new iteration by executing *prologue code* and then assigns it to the newly created thread. After the initialization work, the main thread enters a main loop where it first calls waits for messages from parallel threads and responds accordingly. It processes Exception requests from all parallel threads until finally a Finish message is

```
...
init_version_table();
for(i=0; i<Num_Proc; i++) {
    prologue code;
    create thread i;
    init_thread(i);
}

i=0;  //thread id
for (...;...;...)  {    // spec. parallelized loop
  while (1) {
    use "select" call to monitor every massage buffer;
    if any message m is received in msg_buffer[ j ] {
      if ( m.type == "Exception") {
        place the requested value and version
          number into a reply message;
        send_reply_msg(msg_buffer[j]);
      }
      if (m.type == "Finish"  and i == j) {
          result = finish_thread(i);
          if (result == SUCCESS)  {
                break;
          }
          else {
                update C space for thread i;
                send_start_msg(msg_buffer[i]);
          }
      }
    }
  } // end while

 epilogue code;

 prologue code;   //assigning new work
 init_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;
...
```

(a) Main Thread

```
void init_version_table()  {

 int pos= 0;
 Version Table vt =  malloc();

 for every variable var that is potentially
   read or written by parallel threads {

   vt[pos]. D_addr = var's D address;
   vt[pos]. Ver =0;
   pos++;
 }

}
```

(b) Initializing Version Table

```
void init_thread(int i)  {

 if ( thread i's mappting table
     does not exist) {
   Mapping Table mt[i] =  malloc();
 }
 else {
   Mapping Table mt[i] = thread i's
           mapping table;
 }
 int pos= 0;
 for every variable var that requires
   copy-in operations {
     memcpy (var's P address, var's D
           address, var's size);
     mt[i][pos].D_addr = var's D address;
     mt[i][pos].P_addr = var's P address;
     mt[i][pos].Size = var's size;
     mt[i][pos].Version = var's current
             version in the Version Table;
     mt[i][pos].Write_Flag = 0;
     pos++;
 }
}
```

(c) Initializing Parallel Threads

```
Bool finish_thread(i) {

   int pos1, pos2;
   int flag = True;
   Version Table vt = the main thread's version table
   Mapping Table mt[i]  = thread i's mapping table;

   //check speculation
   for (pos1=0; mt[i][pos1] != NULL; pos1++) {
         pos2 = the position of the entry in vt, where
           entry.D_addr matches mt[i][pos].D_addr;

         if (mt[i][pos1].version != vt[pos2].Ver) {
                 flag = False;
                 break;
         }

   }

   if (flag == False) {
       return FAIL;
   }
   else {
        for (pos1=0; mt[i][pos1]; pos1++) {
          if (mt[i][pos1].Write_Flag) {

            //perform copy-out operations
             memcpy(mt[i][pos1].D_addr,
                 mt[i][pos1].P_addr, mt[i][pos1].Size);

            //update version table
            pos2 = the position of the entry in vt, where
              entry.D_add matches mt[i][pos].D_addr;
               vt[pos2].ver ++;
          }
        }
       return SUCCESS;
   }
}
```

(d) Finishing Parallel Threads

```
void *thread_wrapper(i)
{
   while(1)
   {
     wait_start_msg(msg_buffe[i]);
     speculative body code with
          access checks;
     send_fini_msg(msg_buffer[i]);

   }
}
```

(e) Parallel Threads

```
For each variable var's access:

if (var has not been copied into
   P space) {
     put the D address, size into a
           message m;
     m.type = "Exception";
     send_msg(msg_buffer[i], m);

     //get the value from message
     val = read(msg_buf[i]);
     store val into var's P address;

     //update the mapping table
     int ver = read(msg_buf[i]);
     int pos = empty position in the
         thread i's mapping table mt[i];

     mt[i][pos].D_addr = var's D address;
     mt[i][pos].P_addr = var's P address;
     mt[i][pos].Size = var's size;
     mt[i][pos].Version =  ver;
     mt[i][pos].Write_Flag = 0;
}
```

(f) Access Checks

Fig. 4 Code transformation

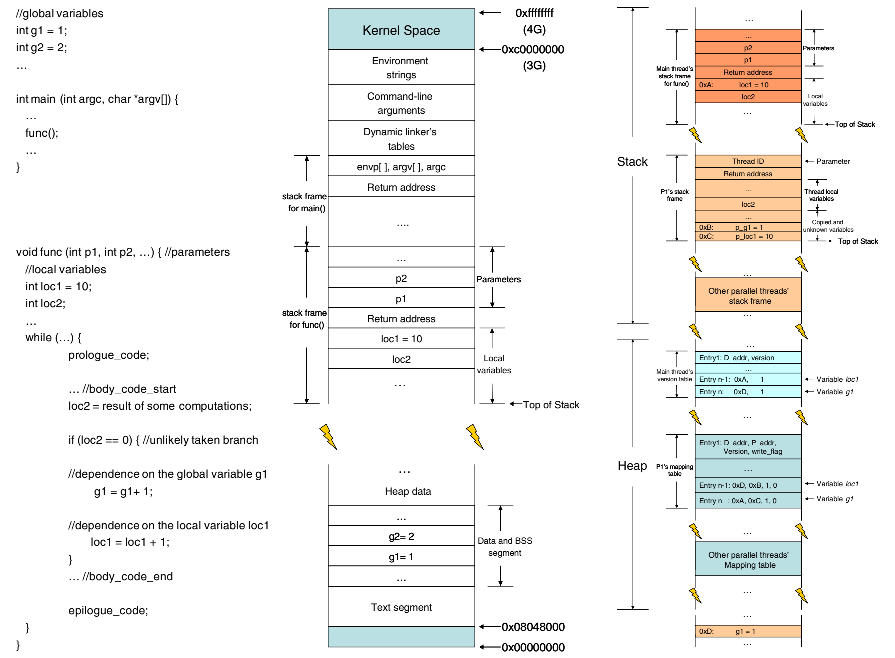
received from the parallel thread that executed the earliest speculative iteration currently assigned to the parallel threads (this is thread `i` in the code). Upon receiving this message, it calls `finish_thread` routine [see Fig. 4d]. This routine first performs checks to detect *misspeculation* by examining the version numbers. If speculation is successful, it commits the results and returns SUCCESS. Otherwise it returns FAIL and as a result the main thread prepares the parallel thread for reexecuting the assigned iteration. Committing result is essentially implemented by performing copy-out operations by consulting the mapping table. 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.

(Parallel Thread) After its creation, each parallel thread executes function `thread_wrapper` shown in Fig. 4e. After entering the `while` loop, a parallel thread `i` waits for the `start` message from the main thread. Upon receiving this message, it will execute the *speculative body code* and then send the Finish message to the main thread. Note that the speculative body code is also obtained by transforming its corresponding sequential code in couple of ways. First, code is introduced to perform updates of the `write_flag` in the mapping table. The above code is put immediately following every store access to variable `p_var` which can be of any type but `thread local`. This information is used by the main thread when the copy-out operations are performed. Second, code for access checks must be introduced. When a `unknown` variable is read, we need to check the if its one-bit tag is true. If not, that means this variable has not been copied into the current thread's P space, so we copy it on the fly. Note that the above code is inserted immediately before every read access to variable `p_var`. Although copying on-the-fly seems to be complex and may potentially slow down the execution due to the use of message passing, it does not cause much overhead at runtime because of two reasons. First, these `unknown` variables are not accessed in the profiling run and hence, are very unlikely to be accessed in the normal run. Therefore, the likelihood of executing the corresponding access checks is very small. Second, the copy operation for each `unknown` variable only needs to be performed once. For the subsequence accesses, we do not have to send any request to the main thread as the value has been copied into the P space of the current thread.

(Other Issues) 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.

3.3 Runtime Memory Layout

The previous section described how to transform the code from sequential version to the parallel version. In this section, we show the execution difference between these two versions via the comparison of the virtual memory layout under the 32-bit linux OS. Figure 5a shows an example of the sequential code with variables needing different types of treatment under our copying scheme. In particular, `g1` and `g2` are global variables. The `main` function calls `func` which has `p1` and `p2` as its parameters. Before the main loop `while`, there are two local variables `loc1` and `loc2` declared in the `func`. The `while` loop has already been partitioned into three parts. Note that in the *body code*, `loc2` is classified as a `thread local` variable as it is always defined before use. An unlikely taken branch is introduced in which `g1` and `loc1` carry out a loop dependency respectively. In other words, we can speculate on these two variables since the two increment statements are unlikely to be executed.



```
//global variables
int g1 = 1;
int g2 = 2;
...

int main (int argc, char *argv[]) {
  ...
  func();
  ...
}


void func (int p1, int p2, ...) { //parameters
  //local variables
  int loc1 = 10;
  int loc2;
  ...
  while (...) {
          prologue_code;

          ... //body_code_start
          loc2 = result of some computations;

          if (loc2 == 0) { //unlikely taken branch

          //dependence on the global variable g1
               g1 = g1+ 1;

          //dependence on the local variable loc1
              loc1 = loc1 + 1;
          }
          ... //body_code_end

          epilogue_code;
  }
}
```

(a) Sequential Code **(b)** Stack Sequential Executior **(c)** Stack of Parallel Executior

Fig. 5 Runtime memory layout for sequential version and parallel version

Figure 5b shows the virtual memory layout when the `func` is being sequentially executed. As we can see, The text segment start from the bottom (0x8048000). Then data and BSS segment where `g1` and `g2` are allocated, are next to it. Above these two segments, heap starts growing towards higher virtual addresses. On the other side, we can see that the address that can be used as the stack is from 0xc0000000 (3G) to lower addresses. All local variables, parameters and the return address of the `func` are stored on the stack. In particular, right after the stack frame for `main`, the parameters of `func` (e.g., `p1` and `p2`) are stored. All local variables of `func` (e.g., `loc1` and `loc2`) are stored after the `return address` location. As the execution continues, the stack will grow and shrink accordingly.

Figure 5c shows the virtual memory layout when the `func` is being executed under the CorD execution model. First, the stack frame for `func` now is split into several frames, one for each thread. The very top frame is for the main thread which is identical to the sequential version. The one next to it is the frame of the first parallel thread `P1`. This frame contains one parameter (`ThreadID`), the return address, thread local variables (e.g., `loc2`), copied variables (e.g., `p_g1` which corresponds to `g1`, `p_loc1` which correspond to `loc1`) and unknown variables. Note that the copied variables have correct values at the start of the parallel thread's execution because of the copy-in operations performed by the main thread. Other threads also have a similar stack frame, which essentially is the P space in the CorD.

The C space in the CorD, on the other hand, is implemented through the `malloc` function call. In other words, the mapping table for each parallel thread and the version table for the main thread are allocated in the heap. From the figure, we can see that each entry contains an address and its current version in the version table. For example, the global variable `g1`'s current version is 1. As we can see, the mapping table (P1's mapping table is shown in the figure) contains the mapping information of each copied variable. For instance, the address 0xD (corresponding to `g1`) is mapped to the address 0xB (corresponding to `p_g1` in P1's stack).

Note that although message buffers are implemented through the `pipe` function call, and thus allocated by OS in the kernel space (not shown in the figure), they still conceptually belong to the C space in our execution model.

3.4 Loop Parallelization Variants

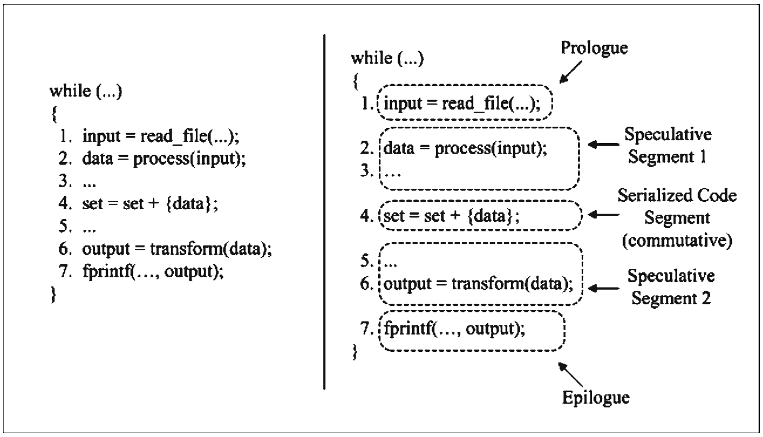
The loop parallelization algorithm developed in this section assigns individual iterations of the loop to separate cores. Once an iteration is completed, its results are committed to non-speculative state. However, there are situations in which it is more appropriate to commit results of executing multiple iterations or results of executing a part of a single iteration to non-speculative state. We discuss these variants of speculative loop parallelization in this section.

(**Committing Results of Multiple Iterations**) The performance of the parallel version may be hindered by *thread idling*. If a parallel thread that is assigned work earlier, finishes its work before some later threads are assigned work by 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. This situation arises when relative to the number of cores being used, the work being assigned to a single core in one step is quite small. Thus, to avoid thread idling, we can increase the work assigned to each parallel thread in a single step, i.e. the main thread can assign two or more iterations of work to a parallel thread in a single step. In this way, we ensure every parallel thread stays busy while the main thread is still assigning the later iterations to other parallel threads.

(Committing Results of a Part of an Iteration) During the partitioning of the loop body our algorithm constructed the speculative body in such a manner that it did not contain any statements that are involved in frequently occurring (or definitely occurring) loop dependencies. This was done because frequent dependences will give rise to frequent misspeculations; hence making the performance of the parallel execution no better than the sequential execution. Currently statements involving these dependences must be included in the prologue of the epilogue. However, we observed that in some programs the inclusion of statements involved in frequent or definite dependences into the prologue or epilogue yielded a very small speculative body. In other words most of the statements are part of the prologue or epilogue and relatively few statements are actually contained in the speculative body. Thus, we cannot expect significant performance enhancement in such situations.

We observe that to address the above problem, and effectively exploit the parallelism available in loops, we must partition a loop iteration into multiple distinct speculative code segments. These speculative code segments are interrupted by statements involving frequent or definite dependences and thus their execution across parallel threads must be serialized. This modified approach yields a parallelized loop in which misspeculations are once again an infrequent occurrence. Another consequence of allowing multiple speculative code segments is that after one such segment is executed its result must be committed before the execution of a parallel thread can proceed further. In other words, misspeculation checks followed by commit or re-execute must be performed multiple times for different speculative segments of the loop body.

Figure 6 shows a loop requiring the partitioning of the loop body such that it contains two distinct speculative code segments. On the left, a *while* loop contains typical input and output statements which can be put into the prologue and epilogue respectively. However, we cannot put the rest of the statements into a single speculative body because of statement 4, which introduces a dependences across all loop iterations. Therefore, we divided statements 2–6 into three parts as shown on the right: the first speculative code segment; the serialized segment; and the second speculative code segment. During execution, when each speculative code segment is started, the main thread performs copy-in operations; and when each speculative code segment is finished, the main thread performs the misspeculation checks. If speculation succeeds, copy-out operations are performed; otherwise, only the failed part is re-executed. The execution of statement 4 that intervenes the execution of two speculative code segments never causes misspeculation as its execution is serialized. We further observe that the executions of statement 4 by different loop iterations being executed in parallel can be carried out in any order, i.e. they are *commutative* [1,2]. Thus, the updates to variable *set* can be expressed and implemented as transactions.



```
while (...)
{
 1. input = read_file(...);
 2. data = process(input);
 3. ...
 4. set = set + {data};
 5. ...
 6. output = transform(data);
 7. fprintf(..., output);
}
```

```
while (...)
{
 1. input = read_file(...);
 2. data = process(input);
 3. ...
 4. set = set + {data};
 5. ...
 6. output = transform(data);
 7. fprintf(..., output);
}
```

Fig. 6 Partitioning a speculative body into several parts

4 Experiments

4.1 Experimental Setup

We have implemented the speculative loop parallelization approach described in this paper and carried out experiments for a set of programs. 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 [3] instrumentation framework to enable profiling. Since the output of Pin is in the form of instruction 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 [4] 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 (i.e., total of 8 cores) 3.0 GHz Xeon machine with 16 GB memory.

The benchmarks used in the experiments are divided into two groups. The first group contains five programs taken from *SPEC* [5] and *MiBench* [6] suites in which the program transformation that creates a single speculative code segment is used, i.e. one loop iteration or multiple loop iterations are assigned to a single parallel thread and after all of the assigned work is finished the results are committed to non-speculative state. The programs in this group include: `197.parser`, `130.li`, `256.bzip2`, `255.vortex`, and `CRC32`. The second group of five programs is taken from the *STAMP* [7] suite of programs and they include `bayes`, `kmeans`, `labyrinth`, `vacation`, and `yada`. The main loops in all these programs contain definite loop dependencies caused by commutative statements. Thus, parallelization in these programs requires creation of multiple speculative code segments from a single loop iteration which are intervened by statements involving definite loop dependences. Note that we only use selected benchmarks from various suites in our experiments. The rest of the programs were not used due to a variety of reasons: they could not be parallelized using our approach, the parallelism is insufficient causing no net benefit in performance, or the program could not be currently handled by our compilation infrastructure.

Table 2 describes the characteristics of the programs we used. In this table, the first two columns show the name and the number of source code lines of each 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.

4.2 Execution Speedups

Next we present the results of execution speedups obtained using our parallelization algorithms. In this experiment, we first measured the baseline which is the sequential execution time of the loop that was parallelized. Then we measured the execution time of this loop in our model with different numbers of parallel threads. Figure 7 shows the speedup for the the first group of programs (*SPEC* and MiBench programs). From Fig. 7, we can see that when the number of parallel threads increases, the speedup for all benchmarks goes up linearly. The highest speedup achieved ranges from 4.1 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.

The data shown in Fig. 7 was obtained by selecting loop parallelization that reduced the effect of thread idling. In particular, with the exception of `CRC32`, thread idling was observed in the four other programs whenever more than 5 parallel threads were created. In fact we observed that when a single iteration of work was assigned to each parallel thread, thread idling caused performance to remain unchanged for `197.parser`, `130.li`, `256.bzip2`, and `255.vortex` when the number of threads was increased beyond 5 threads. However, when multiple iterations were assigned to each parallel thread in a single step, thread idling was eliminated and improvements in speedups was observed. Figure 8a–d shows the speedups of these benchmarks with and without elimination of thread idling.

As we can see from the figure, when less than 5 threads are used, the use of the optimization does not affect the speedup as all parallel threads are busy. However, when more than 5 threads are used, without using the optimization higher speedups

Table 2 Characteristics of benchmarks

Program	LOC (K)	Prof. input	Exp. input	Prologue	Epilogue	Vars. in body			
						I	O	L	IO
197.parser	9.7	1 K file	36 K file	fgets	printf	49	6	12	2
130.li	7.8	6 scripts	72 scripts	i++	printf, var++	30	0	3	6
256.bzip2	2.9	200 K file	7.5 M file	fgetc	fputc	12	8	11	1
255.vortex	49.3	Test/input	Train/input	++i	fprintf	76	5	4	6
CRC32	0.2	One 4M-file	80 4M-files	−−argc	printf	1	0	2	1
Bayes	3.1	Simulator input	Non-simulator input	Get a task	Update TMLIST	4	0	4	1
Kmeans	0.6	n2048-d16-c16	n65536-d32-c16	loop++	Update new center	5	0	3	2
Labyrinth	1.6	x32-y32-z3-n96	x512-y512-z7-n512	Get a task	Update PVECTOR	3	0	2	3
Vacation	1.8	Simulator input	Non-simulator input	i++	None	6	0	5	2
Yada	1.6	633.2	ttimeu1000000.2	Get a task	Update region, var++	2	1	2	4

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

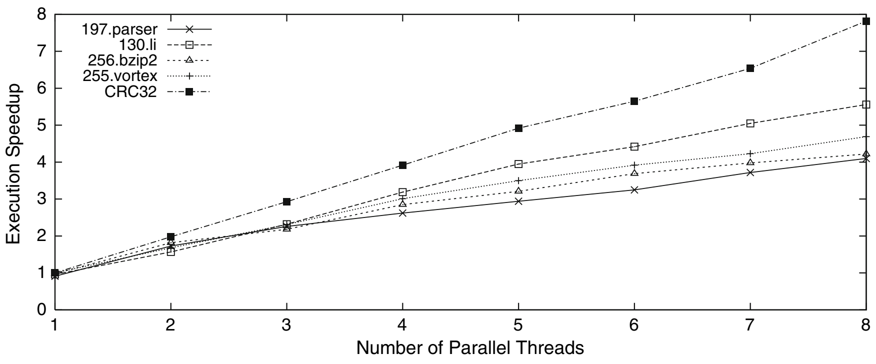


Fig. 7 Execution speedups for SPEC and MiBench programs

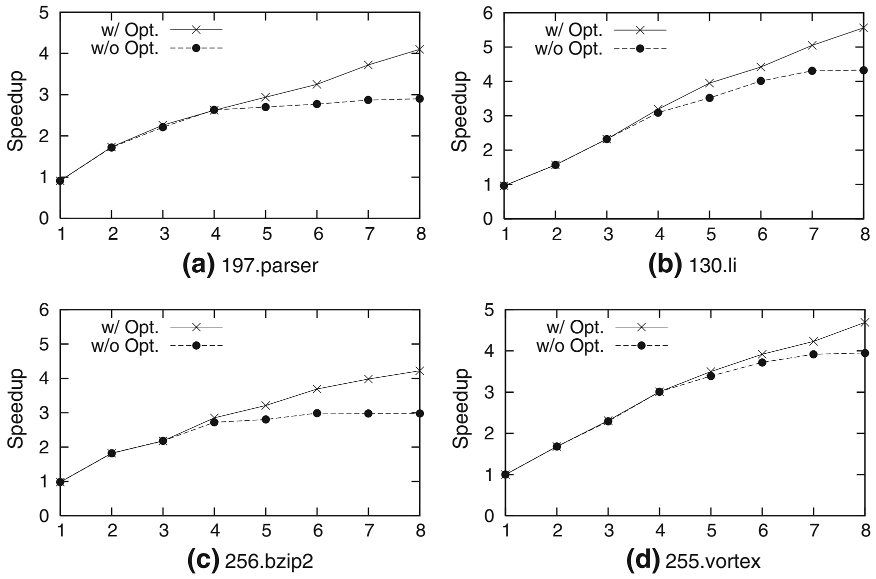


Fig. 8 Handling thread idling

are not possible because the additional threads are simply waiting for their result to be committed and new work to be assigned to them. Note that in the case of `CRC32`, each single iteration processes a 40 MB file which is a big enough computation to keep each thread busy. Therefore, assigning multiple iterations to one thread will not result in any additional speedup.

Figure 9 shows the speedups for the second group of programs (*STAMP* benchmarks). Most of the computations performed in these programs are performed on either the global heap data or the local heap data. The accesses to the global heap are the ones that give rise to serialized code segments inside a loop which are commutative and thus as long as these regions are executed atomically, the correctness will be ensured regardless their execution order. In our implementation of the parallel

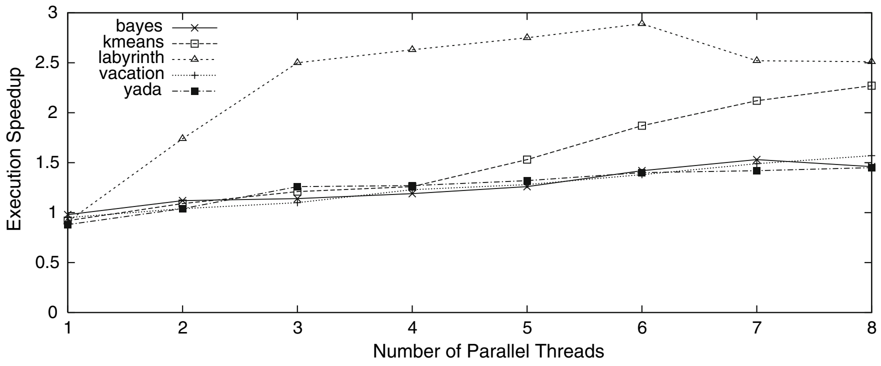


Fig. 9 Execution speedups for STAMP programs

versions, software transactional memory system was used to execute the commutative regions. In this way, our parallel threads do not require the global heap data to be copied into their own P space. Instead, they can directly access the D space memory with the assistance of transactional memory support provided in [8]. The rest of the speculative code segments are executed using our CorD model.

From Fig. 9, we can see that for all 5 programs we used, we obtain some benefit from parallelizing the execution. However, the speedups that result in smaller than those that were observed for the first set of programs. This is because of several reasons. First, in some benchmarks, the non-parallelizable code takes a significant amount of execution time (`bayes`, `yada`). Second, the execution of serialized code limits the speedup that can be obtained. In fact, when 7 or 8 threads are used in program labyrinth, the performance worsens because the serialized code causes significant amount of transaction conflicts which nullifies some of the parallelism benefits. Third, more interactions between the main thread and parallel threads are required due to the partition of the body code, which increases the overhead of using our parallelization strategy.

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 3. 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

Table 3 Overhead breakdown 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
255.vortex	5.28	0.04	0.01	1.25	0.39
CRC32	0.02	0	0	0.01	0.32
Bayes	0.43	0	0	0.15	0.11
Kmeans	0.57	0	0	0.02	0.38
Labyrinth	0.13	0	0	0.01	0.09
Vacation	0.89	0	0	0.66	0.30
Yada	0.87	0	0	0.53	0.26

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 SPEC benchmarks and less than 1% instructions for `CRC32` and STAMP benchmarks, which do 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.

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 Fig. 10. The space used by the sequential version serves as the baseline.

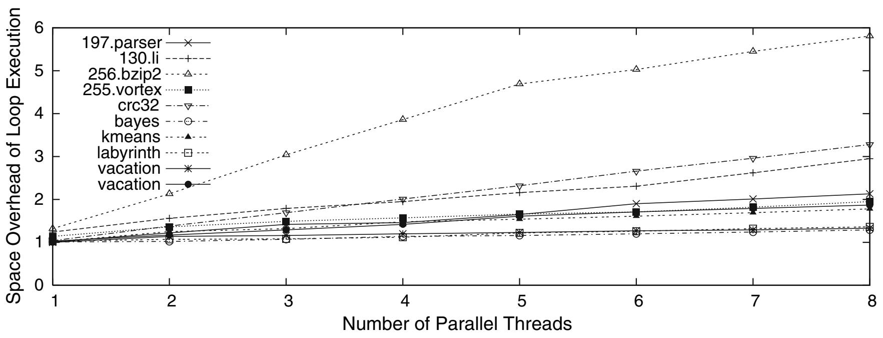


Fig. 10 Memory overhead

Table 4 Size of binary code

Program	Sequential version (K)	Parallel version (K)
197.parser	234	239
130.li	179	183
256.bzip2	53	57
255.vortex	1336	1370
CRC32	8	10
Bayes	165	170
Kmeans	47	48
Labyrinth	106	111
Vacation	170	173
Yada	182	186

As we can see, the overhead for most benchmarks is between 1.3x and 3.2x 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.

Besides the dynamic space consumption, we also examined the increase in the static size of the binary. As shown in Table 4, the increase varied from 1 K to 5 K for most programs, a very small fraction of the binary size.

4.4 Comparison with Other Techniques

A significant amount of research has been performed on thread level speculation (TLS) [9–15]. Similar to our work, TLS is aimed at extracting parallelism from sequential codes. However, TLS relies heavily upon hardware support for executing threads speculatively. In particular, TLS requires a multithreading processor which has the ability to detect misspeculation and recover the computation state. Thread isolation is achieved by either versioned cache or versioned memory. These hardware features are expensive and not available in commercial processors at this time.

In comparison to TLS techniques, the techniques in [16] and this paper are implemented purely in software. There is no requirement for hardware to detect misspeculation or isolate speculative state. Hence, these software solutions are practical for use on machines available today. In this we demonstrate the advantages of our technique over the technique in [16] through experimentation. Note that in Ding et al. [16], proposed a *process based* runtime model that enables speculative parallel execution of Potentially Parallel Regions (PPRs) on multiple cores. We compare our work to this process based work in three respects: overall performance, copying overhead, and process creation overhead.

Figure 11 shows the comparison of speedups obtained by our technique and estimated speedups of the technique in [16]. The comparison is made for all benchmarks

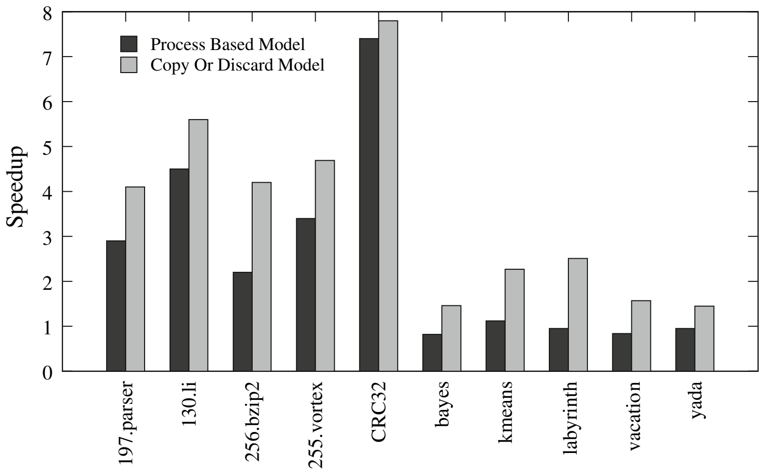


Fig. 11 Speedup comparison

when 8 processes/threads are used. As we can see, our approach outperforms the process based model proposed in [16] across all benchmarks. In particular, for the first group of benchmarks, excluding CRC32, the performance difference ranges from 1.1 for 130.li to 2.0 for 256.bzip. In case of CRC32 the performance difference is smaller – 0.2. For the second of group of benchmarks, while our approach can achieve 1.4x–2.5x speedup, the process based model slows down the original execution in all programs except kemeans where 1.12x speedup is achieved.

There are three reasons for the above performance difference. First, in [16], the speculative region is executed by a process instead of a thread. Each process can only communicate with its child process, and the last process cannot know the termination of the first process if more than two processes are created. Therefore, the work has to be assigned to the processes running on different cores in rounds. Thus, parallelism cannot be fully exploited.

Second, copying overhead for [16] is large. As already mentioned, in [16], the speculative region is executed by a process instead of a thread. The advantage of using a process is that all required data by a speculative process is supplied by the OS through copy-on-write scheme. While this makes implementation of the runtime system easy, the copying overhead is higher as copying at OS level is carried out at the granularity of a page. In particular, once a memory cell is written in a speculative process, OS will make a copy of the entire page containing that cell. It is worth noting that more pages need to be allocated so as to solve the false sharing problem caused by tracking the dependence at page level. According to the description in [16], each global variable needs to be allocated on a distinct page. This worsens the overhead of the copying operation. In contrast, a thread in our model only copies the data that can be potentially accessed. This will avoid a large amount of unnecessary copying operations.

Table 5 shows the average size of the data that needs to be copied for each working process or thread in the two approaches. As we can see, while the size of data copied in our approach ranges from 0.1 KB to 965 KB across all benchmarks, it ranges from

Table 5 Copying overhead comparison

Program	Copy-on-discard model (K)	Process-based model
197.parser	12	4.6 M
130.li	6	5.5 M
256.bzip2	965	17.9 M
255.vortex	19	4.1 M
CRC32	0.1	395 K
Bayes	56	34.8 M
Kmeans	67	58.7 M
Labyrinth	2	36.6 M
Vacation	62	25.6 M
Yada	27	17.9 M

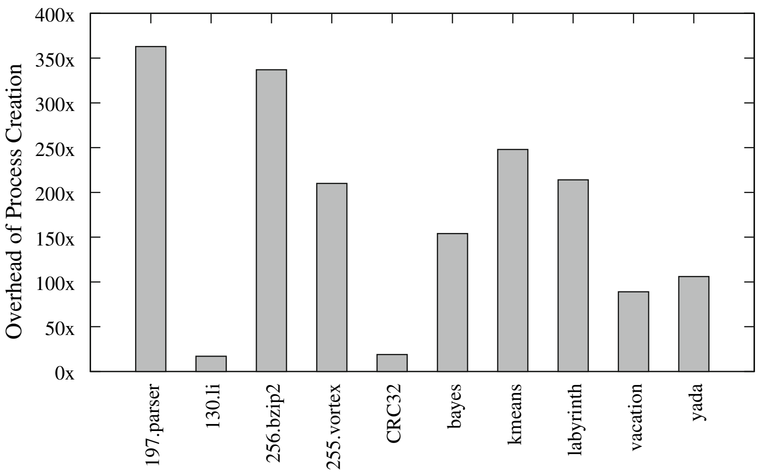


Fig. 12 Overhead of process creation per core

395 KB to 58.7 MB for process based approach. In particular, for each benchmark, the data copied in the process based model is much more than in our approach. Clearly, the time for copying the data is significant.

Lastly, the overhead of process creation is high. In [16] new processes are continuously created when old processes finish their execution. In the case a loop has large number of iterations, frequently creating processes negatively impacts performance. In contrast, our work creates working threads only once. After the creation, each thread simply waits for the main thread to dispatch a task to it. A thread terminates only after the entire loop has been executed. To show the difference in these costs, we measured the average time of process creation across all 8 cores for the process based model. We also measured the time of creating 8 threads in our model. The ratio of process creation time to the thread creation time is shown in Fig. 12.

As we can see, the ratio is between 17 and 363, which means that the process based model spent much more time on creating processes on each core than our model where totally 8 threads are only created once. Note that the process creation time varies on each benchmark because it is directly affected by the number of speculative tasks. The more the number of tasks is, the greater the number of processes created.

5 Related Work

Ding et al. [16] proposed a *process based* runtime model that enables speculative parallel execution of PPRs on multiple cores. However, 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. Besides, due to use of processes, significant amount of memory pages need to be copied during the speculative execution. Moreover, each parallel region has to be executed in one process, and hence processes have to continuously created. None of the above drawbacks are present in our approach. Kulkarni et al. [2,17] 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.

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) [18–20] 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 [21–25].

The alternative approach to exploiting loop parallelism is DO-ALL technique [2,9–17] where each iteration of a loop is executed on one processor. Among these works, a large number of them focus on TLS which essentially is a hardware-based technique for extracting parallelism from sequential codes [9–15]. 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. [26] 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. [27] 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.

There has been significant research work on transactional memory systems [28–31]. Although it has some similarities with out speculative parallelization work, there are some fundamental differences. First, the purpose of developing transactional memory is to replace the error-prone synchronization locks. Its main capability is to ensure atomicity. Therefore, it appears in the context of developing concurrent programs and used by programmers. Our work, on the other hand, focuses on the sequential loop parallelization, so the parallel version can use transactional memory to maintain the atomicity property if necessary. Second, the implementation of a transactional system requires tracking every memory cell. However, our approach tracks every variable thanks to the use of profiling and compiler technique. Thus, the overhead of our technique is smaller. Moreover, the compiler performs program specific optimizations during the compilation while the transactional memory is not capable of that.

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 significant speedups on a server with two Intel Xeon quad-core processors.

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

References

1. Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T., August, D.: Revisiting the sequential programming model for multi-core. In: MICRO, pp. 69–84 (2007)

2. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Paul Chew, L.: Optimistic parallelism requires abstractions. In: PLDI, pp. 211–222 (2007)
3. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI, pp. 190–200 (2005)
4. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88 (2004)
5. http://www.spec.org
6. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: a free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization (2001)
7. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: stanford transactional applications for multi-processing. In: IISWC, pp. 35–46 (2008)
8. Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: DISC, pp. 194–208 (2006)
9. Cintra, M.H., Martínez, J.F., Torrellas, J.: Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: ISCA, pp. 13–24 (2000)
10. Hammond, L., Willey, M., Olukotun, K.: Data speculation support for a chip multiprocessor. In: ASPLOS, pp. 58–69 (1998)
11. Vijaykumar, T.N., Gopal, S., Smith, J.E., Sohi, G.S.: Speculative versioning cache. IEEE Trans. Parallel Distrib. Syst. **12**(12), 1305–1317 (2001)
12. Gregory Steffan, J., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: ISCA, pp. 1–12 (2000)
13. Bhowmik, A., Franklin, M.: A general compiler framework for speculative multithreading. In: SPAA, pp. 99–108 (2002)
14. Marcuello, P., González, A.: Clustered speculative multithreaded processors. In: ICS, pp. 365–372 (1999)
15. Zilles, C., Sohi, G.: Master/slave speculative parallelization. In: MICRO, pp. 85–96 (2002)
16. Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., Zhang, C.: Software behavior oriented parallelization. In: PLDI, pp. 223–234 (2007)
17. Kulkarni, M., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Paul Chew, L.: Optimistic parallelism benefits from data partitioning. In: ASPLOS, pp. 233–243 (2008)
18. Ottoni, G., Rangan, R., Stoler, A., August, D.I.: Automatic thread extraction with decoupled software pipelining. In: MICRO, pp. 105–118 (2005)
19. Raman, E., Ottoni, G., Raman, A., Bridges, M.J., August, D.I.: Parallel-stage decoupled software pipelining. In: CGO, pp. 114–123 (2008)
20. Vachharajani, N., Rangan, R., Raman, E., Bridges, M,J., Ottoni, G., August, D.I.: Speculative decoupled software pipelining. In: PACT, pp. 49–59 (2007)
21. Buck, I.: Stream computing on graphics hardware. PhD thesis, Stanford, CA, USA (2005)
22. Fan, K., Park, H., Kudlur, M., Mahlke, S.A.: Modulo scheduling for highly customized datapaths to increase hardware reusability. In: CGO, pp. 124–133 (2008)
23. Kapasi, U.J., Rixner, S., Dally, W.J., Khailany, B., Ahn, J.H., Mattson, P., Owens, J.D.: Programmable stream processors. Computer **36**(8), 54–62 (2003)
24. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. In: PLDI, pp. 114–124 (2008)
25. Thies, W., Chandrasekhar, V., Amarasinghe, S.: A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In: MICRO, pp. 356–369 (2007)
26. Vijaykumar, T.N., Sohi, G.S.: Task selection for a multiscalar processor. In: MICRO, pp. 81–92 (1998)
27. Chu, M., Ravindran, R., Mahlke, S.: Data access partitioning for fine-grain parallelism on multicore architectures. In: MICRO, pp. 369–380 (2007)
28. Adl-Tabatabai, A.-R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI, pp. 26–37 (2006)
29. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: ASPLOS, pp. 336–346 (2006)
30. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA, pp. 289–300 (1993)
31. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting nested transactional memory in logtm. SIGOPS Oper. Syst. Rev. **40**(5), 359–370 (2006)