Enhanced Speculative Parallelization Via Incremental Recovery

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

Abstract

The widespread availability of multicore systems has led to an increased interest in speculative parallelization of sequential programs using software-based thread level speculation. Many of the proposed techniques are implemented via *state separation* where non-speculative computation state is maintained separately from the speculative state of threads performing speculative computations. If speculation is successful, the results from speculative state are committed to non-speculative state. However, upon misspeculation, *discard-all* scheme is employed in which speculatively computed results of a thread are discarded and the computation is performed again. While this scheme is simple to implement, one disadvantage of *discard-all* is its inability to tolerate high misspeculation rates due to its high runtime overhead. Thus, it is not suitable for use in applications where misspeculation rates are input dependent and therefore may reach high levels.

In this paper we develop an approach for *incremental recovery* in which, instead of discarding all of the results and reexecuting the speculative computation in its entirety, the computation is restarted from the earliest point at which a misspeculation causing value is read. This approach has two advantages. First, the cost of recovery is reduced as only part of the computation is reexecuted. Second, since recovery takes less time, the likelihood of future misspeculations is reduced. We design and implement a strategy for implementing incremental recovery that allows results of partial computations to be efficiently saved and reused. For a set of programs where misspeculation rate is input dependent, our experiments show that with inputs that result in misspeculation rates of around 40% and 80%, applying incremental recovery technique results in 1.2x-3.3x and 2.0x-6.6x speedups respectively over the discard-all recovery scheme. Furthermore, misspeculations observed during discard-all scheme are reduced when incremental recovery is employed – reductions range from 10% to 85%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Performance, Languages, Design, Experimentation

Keywords Speculative Parallelization, Incremental Recovery, Multicore Processors

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

1. Introduction

Exploiting loop level parallelism is critical to improving the performance of sequential applications on multicore processors. However, many sequential loops cannot be parallelized via static analysis based parallelization techniques due to the presence of cross-iteration dependences. However, it has been observed that in many applications such dependences, although potentially present, rarely manifest themselves at runtime and thus despite the presence of cross-iteration dependences these loops are a significant source of parallelism [5, 15, 37]. To exploit such parallelism, speculative parallelization has been proposed that employs *thread level speculation* (TLS) to optimistically extract and exploit parallelism. The compiler first assumes the absence of cross-iteration dependences so that the sequential loops can be optimistically parallelized. When the speculatively parallelized program is executed, the runtime system or specialized hardware is used to detect misspeculation (i.e., manifestation of ignored dependences) and recover from it. While hardware based TLS has been extensively researched, the specialized hardware structures (e.g., versioning cache [8], versioning memory [7] etc.) on which such techniques rely have not been incorporated in commercial multicore processors. On the other hand, software-based TLS has also drawn attention of researchers and the advantage of this approach is that it can be used to take advantage of multicore systems available today.

Software TLS uses runtime systems to detect misspeculations and recover from them. However, the overhead imposed by the runtime system must be reduced so that benefits of parallelism can be realized. Many of the prior works on software TLS rely on *state separation* where non-speculative computation state is maintained separately from speculative state of threads that perform speculative computation and thus recovery from misspeculation can be performed by simply *discarding* speculative state [5, 13, 35–38]. All of these works have demonstrated the benefits of software TLS for applications in which misspeculation events are infrequent (typically a few percent). As misspeculation rate rises, the benefits of parallelism quickly evaporate due to the high recovery cost. This is a serious problem in context of applications where the misspeculation rate is *input dependent* as illustrated next.

We identified five applications in which the misspeculation rate is input dependent. Figure 1 shows the performance of these five programs for inputs which result in low (around 1%), medium (around 40%), and high (around 80%) misspeculation rates. The speedups are measured in comparison to execution time of the sequential version of the application and it is for the case where 4 cores are available for executing speculative threads. As we can see, for medium and high misspeculation rates mostly execution slow down is observed (i.e., speedup is less than 1). While prior works have applied speculative parallelization to these applications, they have only considered inputs with low misspeculation rates to illustrate the benefits. However, since the performance is a function of input characteristics, it is unclear as to whether or not employing

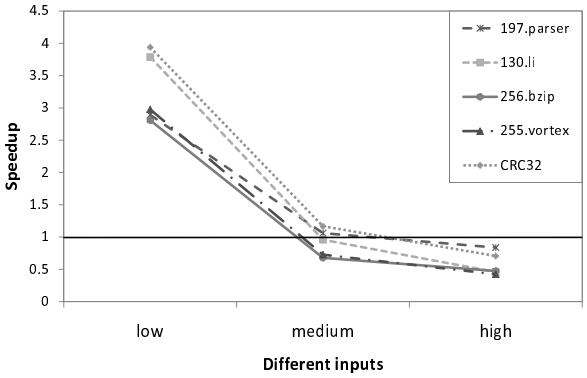

Figure 1. Misspeculation Rate vs. Performance.

speculative parallelization would result in benefit or loss. To overcome this problem, in this paper, we develop an approach which performs well even when misspeculation rates are medium to high. Thus, speculative parallelization can be applied no matter the nature of inputs.

The fundamental reason for the performance loss is that when a misspeculation occurs, all speculatively computed results are assumed to be incorrect and hence discarded. Thus, recovery is expensive as the speculative computation must be repeated in its entirety. We will refer to this strategy as *discard-all*. Some other works employ rollback to recover from misspeculation [15, 18, 21]. The rollback cost is minimized by having the programmer exploit application specific knowledge in providing the rollback code. While state separation has been fully automated, the rollback approach relies on the programmer to provide rollback code. *The goal of this paper is to extend the fully automated approach of state separation so that it can handle high misspeculation rates.*

We have observed that although *discard-all* strategy is simple to implement, it is also wasteful. To recover from misspeculation, it is often necessary to only reexecute part of the computation. This is because while many values may be speculatively read by the speculative computation, the misspeculation may be caused by only some or even by only one of these values. Therefore a significant part of the computation need not be reexecuted, only part of the computation that is dependent on the misspeculation causing value needs to be rexecuted. Based on this observation, this paper presents an approach for *Incremental Recovery* that mitigates the performance loss caused by misspeculation recovery. The *incremental recovery* strategy has two advantages. First, the cost of recovery is reduced as only part of the computation is reexecuted. Second, since recovery takes less time, the likelihood of future misspeculations is reduced. While the benefits of *incremental recovery* are clear, designing an efficient implementation of this strategy is a challenge. We develop an implementation of *incremental recovery* as a natural extension of state separation which allows results of partial computations to be efficiently saved and reused. Our experiments show that *incremental recovery* often yields speedups where *discard-all* produces slowdowns. The key contributions of this work are:

- We demonstrate that there are two benefits of incremental recovery. First, the cost of recovery is lower as only part of the speculative computation needs to be reexecuted when a misspeculation is encountered. Second, faster recovery reduces the likelihood of future misspeculations. We observe that both these factors play a significant role in performance improvement.
- We present the design and implementation of an efficient software based incremental recovery algorithm that allows partial results of a speculative computation to be saved and reused upon misspeculation. The key feature of this implementation is that it simply extends the use of state separation that is already used to implement thread level speculation.
- For a set of programs, our experiments show that even with inputs that result in high misspeculation rates of around 40% and 80%, applying incremental recovery technique can achieve 1.2x-3.3x and 2.0x-6.6x speedups respectively in comparison to the discard-all recovery scheme. Misspeculations during discard-all are reduced by 10% to 85% by incremental recovery.

The rest of the paper is organized as follows. Section 2 describes background on state separation based thread level speculation. Section 3 provides an overview of our approach for incremental recovery. Section 4 develops an efficient implementation of incremental recovery. The experimental results are presented in Section 5. Sections 6 and 7 discuss related work and present conclusions.

2. Background

State Separation. Many prior research works have used State Separation for realizing software based TLS (SS-TLS) [5, 13, 35–38]. State separation causes the non-speculative state of the computation to be maintained separately from the speculative states of parallel computations being performed speculatively (e.g., loop iterations). The first implementation of state separation proposed in [5, 13] creates processes to perform speculative computations – since each process has its own address space, state separation is achieved. Another implementation of state separation, CorD, was proposed by us in [37] and further used in [35, 36, 38]. In this implementation multiple threads that operate is separate portions of the application's address space are used. In addition to parallel threads that perform speculative computations, a main thread is used to coordinate their activities. In particular, the main thread assigns speculative computations to parallel threads, speculatively *copies-in* data into the memories allocated to parallel threads, checks for misspeculations, and if no misspeculation occurs it *copies-out* the speculatively computed results from memories of parallel threads to the memory allocated to hold the non-speculative computation state. Figure 2 summarizes how state separation is achieved in CorD. In this paper we build upon the thread based CorD model because thread based parallelization involves less runtime overhead than process based implementation of state separation.

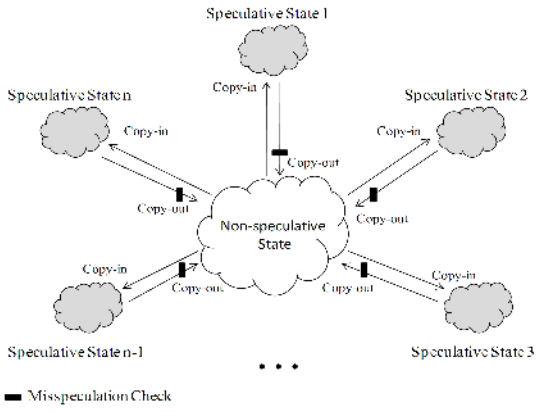

Figure 2. Computation State Under SS-TLS.

Discard-all Recovery. The advantage of using state separation to support speculative execution is the low overhead of handling misspeculations. Since each thread has its own space, updates to the same variable by different threads are performed on its copies in

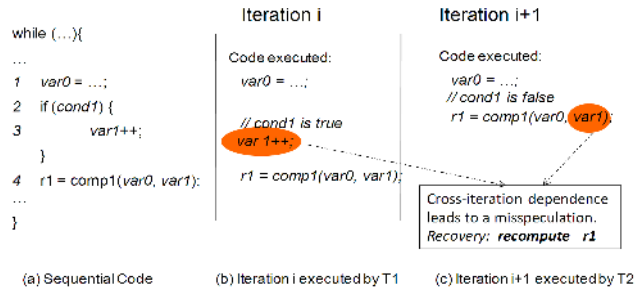

Figure 3. An Example Of Misspeculation Detection And Recovery Using Discard-all In SS-TLS.

different spaces, and thus non-speculative state memory is not affected by any of the parallel threads. Once a misspeculation is detected, the speculatively computed results can be simply discarded and recomputed. Figure 3 illustrates the above using an example. In Figure 3(a) we can see that the sequential loop has a possible cross-iteration dependence due to variable *var1* (dependence is between lines 3 and 4). Suppose this loop is speculatively parallelized and executed by two threads T1 and T2. If *var1* is updated in iteration *i*, the value of *var1* used by T2, which is executing iteration *i+1*, is wrong, because T2 optimistically assumes no cross-iteration dependences will manifest themselves. This dependence leads to a misspeculation and can be captured by the SS-TLS runtime system. To recover from this misspeculation, the system only needs to ask T2 to *discard* the value of *r1* that was speculatively computed, and now non-speculatively reexecute the entire iteration with the latest value of *var1*.

3. Incremental Recovery

Motivation. In SS-TLS, recovery from misspeculations is achieved by discarding all speculatively computed results. This process is shown in Figure 4. Before starting speculative execution of a computation (e.g., a loop iteration), the input values or *live-ins* needed for its execution are speculatively read from non-speculative state and copied into the speculative state for the thread. The reads of values $(v_1, v_2, v_3, \ldots v_n)$ are speculative because the values are read while execution of one or more earlier iterations is in progress; if any of these values is changed by these earlier iterations, then misspeculation occurs. After reading, the computation is performed and then the misspeculation check is executed. If the check fails, misspeculation occurs. To recover from misspeculation, the values of *live-ins* are now read again, this time non-speculatively, and the entire computation is repeated. This strategy for recovery is called *discard-all* as all results computed are discarded and computation is performed again in its entirety. It should be noted that even if the misspeculation occurs due to *any* one of the live-ins, *all* results are discarded. While this approach is simple to implement, discarding all speculatively computed results is a suboptimal solution. It is possible that a subset of speculatively computed results may be correct and thus there may not be a need to perform the entire computation again. Discarding all results can be very wasteful, especially when the misspeculation rate is relatively high because the cost of recovery begins to add up. However, under the above model we lack the ability to identify the part of the computation that need not be reexecuted.

It is worth noting that one update by a thread may cause multiple misspeculations. If *n* parallel threads are simultaneously executing *n* consecutive loop iterations, then misspeculation can occur in *n-1* threads due to a cross-iteration dependence between the first iteration and *n-1* later iterations. In other words, the first thread may update the value of a variable that acts as a live-in variable

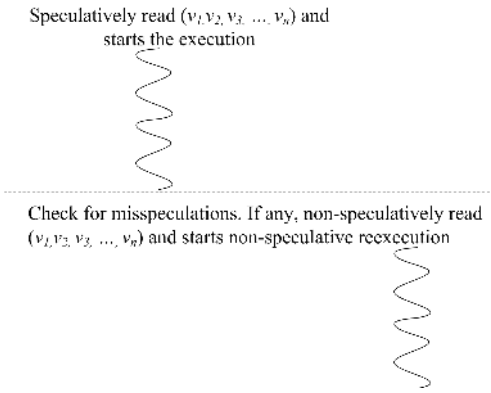

Figure 4. Discard-all Recovery in SS-TLS.

for *n-1* later iterations. This is another reason why the overhead of misspeculation can accumulate to significantly high levels leading to net slowdowns from parallelization as opposed to speedups.

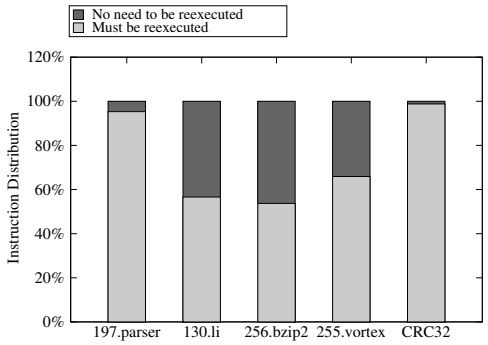

Figure 5. Wasteful Work by Discard-all Recovery.

Figure 5 shows the total waste during the recoveries from a misspeculations for five programs. The waste is measured in terms of the number of instructions. In particular, when a live-in variable is updated in an iteration, we measure the average percentage of instructions that are directly or indirectly dependent upon the updated value within the next iteration. Since an update of a live-in variable usually leads to misspeculations, these instructions have to be reexecuted during the recovery. The remainder of the instructions are the ones that do not depend upon any updated value. Therefore, there is no need to reexecute them. From the figure, we can see that over 40% instructions are not affected when a live-in variable is updated in the preceding iteration. In the case of `197.parser` and `CRC32`, less than 5% instructions are affected. This means the results calculated by more than 95% instructions are still correct even if a cross-iteration dependence arises and it is unnecessary to reexecute the entire iteration.

Our Approach. Motivated by the above study we explore the design of an incremental recovery scheme. There are two key characteristics that must be considered in designing a recovery scheme:

1. First is the *granularity* at which reuse is supported – the finer the granularity the greater is the amount of reuse that can be achieved; and
2. Second is the *complexity* of the scheme that identifies what subset of computation can be reused during recovery.

As the granularity becomes finer, the degree of reuse increases but so does the complexity of identifying subcomputation that

can be reused. An incremental recovery scheme that works at the granularity of a single instruction will provide maximum reuse but will have the highest complexity. On the other extreme is the *discard-all* scheme which the simplest but does not allow any reuse.

Next we present our approach that balances the granularity and complexity. Therefore it allows significant amount of computation to be reused during recovery while at the same time it lends itself to an efficient implementation. As shown in Figure 6, we delineate the computation into many sections according to the points at which the *earliest reads* of the *live-ins* are encountered. In the figure we assume that the first speculative read of v_i appears before the first speculative read of v_{i+1} for all i. Therefore, the live-ins $(v_1, v_2, \cdots v_n)$ cause the computation to be divided into $n + 1$ sections. Now let us assume that of all the values among $(v_1, v_2, \cdots v_n)$ that cause misspeculation, v_i is the one that is read the earliest. In this case we can be sure that the entire computation performed by the sections of code preceding the first read of v_i can be reused. Thus, the recovery can be performed by simply repeating the execution of code starting from the point at which v_i was first read. If v_i happens to be v_1, the reuse achieved is minimum. On the other hand, if v_i happens to be v_n, the reuse is the maximum.

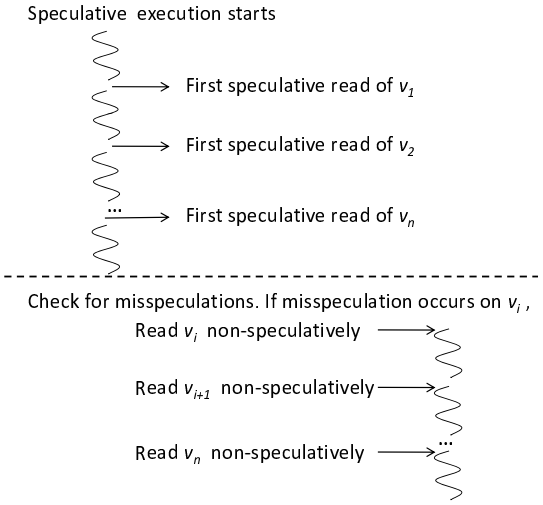

Figure 6. Incremental Recovery in SS-TLS.

To implement the above strategy, we must address the problem of implementing *computation reuse*. The approach we take restores the speculative state of a parallel thread to the point at which it must begin execution during recovery. For example, in Figure 6, the speculative state must be restored to the point that immediately precedes the instruction that represents the *first read* of v_i. We can implement this restoration simply by using the capability of state separation that is already available as follows. At each point that represents a potential first read of a speculative value, i.e. a value from $(v_1, v_2, \ldots v_n)$, we create a new version of the speculative state. In other words, the speculatively computed results generated by each of the executed sections of code in Figure 6 are separated from each other. The misspeculation check is performed in the order the first reads are performed and the first read that causes misspeculation is found. The state prior to this point is available and serves as the basis for performing execution of recovery code.

Thus the key idea we use is to decouple the creation of a parallel thread that performs speculative execution from the creation of the speculative space that it uses for saving results. At each point which represents a first read of a speculative value, a *new version* or *copy* of the speculative space is created. Figure 7 illustrates the idea. Figure 7 shows a parallel thread that creates three speculative spaces

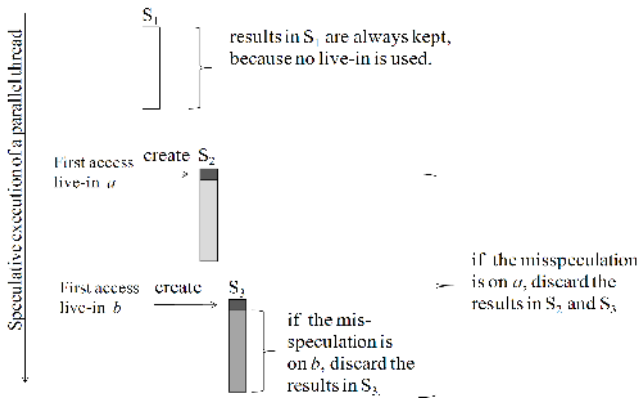

Figure 7. Decoupling Space Allocation From Thread Creation.

S_1, S_2, and S_3 instead of just one space during the speculative execution. Space S_2 is created when the first access of live-in variable a is encountered, and S_3 is created when the first access of live-in variable b is encountered. If a misspeculation is found to have occurred due to a, the results stored in S_2 and S_3 are discarded and recomputed using the correct value of a. However, if the misspeculation is caused by b, only the results in S_3 are recomputed. The results in S_1 are only computed once and are never discarded during recovery because the computation does not use any speculative variable.

Impact of Incremental Recovery on Misspeculation Rate. So far we have discussed one direct benefit of incremental recovery, i.e. reusing previously performed computation to reduce the code reexecuted during recovery. Besides this benefit, the incremental recovery technique has another advantage – it reduces the misspeculation rate. In SS-TLS, variables modified by a parallel thread are stored in the corresponding speculative state. Prior works have shown that copy-in operation of a variable should be performed when the variable is about to be modified, since it can avoid unnecessary copying [36]. If the access to a variable is a read, then the parallel thread can directly read the value from non-speculative state. This is known as copy-on-write scheme, and has been discussed and used in most existing SS-TLS systems [5, 36, 37]. When such scheme is combined with our incremental recovery technique, misspeculation rates can be reduced. Figure 8 illustrates the reason.

Figure 8(a) shows the original recovery scheme. Suppose two iterations *i* and *j* are executed in parallel and variable *var* is speculatively updated in iteration *i* and used in *j*. Further assume the update to *var* is incorrect due to the presence of some other cross-iteration dependence (step 1). As a result, a misspeculation is detected for the thread that is executing iteration *i* (step 2). To recover, this thread has to recompute everything including the value of *var* (step 3) and then commit the correct results (step 5). If the value of *var* is ever read by the thread executing iteration *j* before the results of iteration *i* are committed (step 4), then a misspeculation certainly occurs (step 6).

Figure 8(b) shows the situation where incremental recovery is used. The thread executing iteration *i* still encounters a misspeculation and has to recompute the value of *var*. However, during the recovery if we only recompute the values that were incorrectly computed earlier, then the recovery time is smaller and the results can be committed much faster (step 4). Consequently, the chance of reading the correct value of *var* in a later iteration is significantly increased (step 5) and the misspeculation that had happened before can now be avoided as shown (step 6).

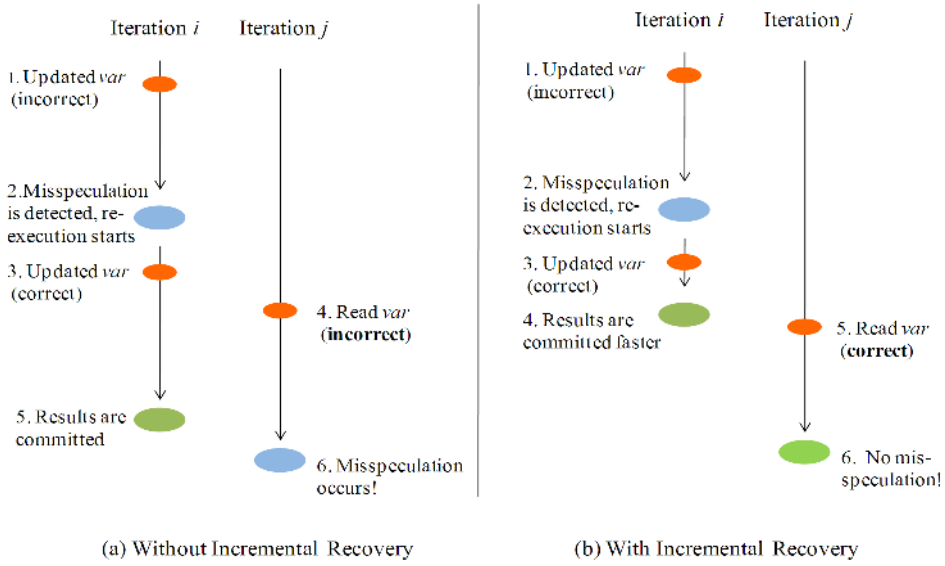

Figure 8. The Effect Of Reducing Misspeculation Rate.

4. Realization of Incremental Recovery

4.1 Creating Multiple Subspaces

Previous section shows that to decouple the space allocation from thread creation, a new subspace must be created when a speculative value is first read. There are different ways to create subspaces. One simple scheme is to dynamically allocate a contiguous memory region (e.g., through *malloc* call) as shown in Figure 9(a). However, repeated allocation of subspaces may cause allocation to eventually fail when the size of subspace is very large. This is because the memory allocator may not find a free memory chunk of the requested size. This situation gets worse with the increase in the number of parallel threads.

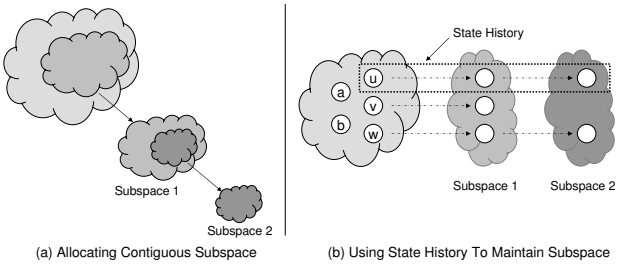

Figure 9. Allocating A Subspace.

To address this problem, we propose to maintain subspaces by using the *state history* for each variable. In particular, the state of an individual variable in a particular subspace is represented as a pair of a *space identifier* (SID) and the variable's value. A SID is a unique integer assigned to each subspace created by the same thread. The states of each variable are linked together in ascending order of space creation time as shown in Figure 9(b). With this scheme, each subspace is not physically contiguous any more. Updating a variable in a different subspace can be simply implemented by adding a (SID, value) pair into this variable's state history. For example, in Figure 9(b), variable *u* and *w* have two different values in subspaces 1 and 2 respectively while *v* is only updated in subspace 1. Thus, subspace 1 is the one that contains all the three variables, subspace 2 only contains *u* and *w*. Since the values of variables *a* and *b* are not affected by speculative values, these variables do not exist in any subspace. The benefit of this approach is that the memory allocated is greatly reduced. Moreover, the memory allocation requests are now at the granularity of variable's size and hence more variables can be handled.

Since each subspace is identified by a SID, each thread maintains a thread local variable *current* SID (CSID) to represent the latest subspace that has been created. When a speculative value is first read, a new subspace is created. The creation can be simply implemented as an increment of CSID. All variables that are defined directly or indirectly using a speculative value are copied into the new subspace. These operations are implemented through live-in access checks and statement transformations as shown in Figure 10.

```
CSID: the current space ID, initialized to 0;
stmt: a statement dst = src1 op src2 executed by thread T

Insert the following live-in access check code before the stmt :
For every source operand s in stmt:
1:  if (s is a live-in variable and is first read by T) {
2:      saving CSID and the PC of the statement for recovery;
3:      CSID++;
4:      sid = CSID;
5:      val = the value of s in D space;
6:      append (sid, val) into the state history of s;
7:  }

To ensure dst is updated in the current subspace, stmt is transformed
into the following:
8:  sid = GetLastSpaceID(dst);
9:  val = GetLastValue(src1) op GetLastValue(src2);
10: if (sid != CSID){ // copy is required
11:     sid = CSID;
13:     append (sid, val) into the state history of dst;
14: }
15: else {
16:     update the last state history record of dst to (sid, val);
17: }
```

Figure 10. Live-in Access Checks And Statement Transformation.

From Figure 10, one can see that the live-in access check code is inserted preceding each statement. For each source operand that

is a live-in variable and is first read by a speculative thread T (line 1), a new subspace is created by incrementing *CSID* by 1 (line 3). As a result, each subspace corresponds to a speculative use of one live-in variable, and later created space has a larger space ID. The old CSID and the PC of current statement are stored and are used if the recovery is needed at this point (line 2). The live-in variable is also copied into this new subspace by appending the pair of CSID and the variable's value in the non-speculative state into its state history (lines 4-6). Thus, the ID of the subspace created for a live-in variable and the speculatively-read value of the live-in are always stored in the first record of this live-in variable's state history.

To update the destination operand *dst*, its subspace ID is first computed (line 8) and the updated value is computed using the latest values of its sources operands (line 9). Since the update has to be performed on the current subspace, the space ID of *dst* is compared with CSID (line 10). If they are different, the *(sid, val)* pair is appended into *dst*'s state history. Otherwise, the last record in the state history is updated with the new pair *(sid, val)*. Function *GetLastSpaceID()* and *GetLastValue()* are auxiliary functions which retrieve the space ID and value respectively from the last record in a variable's state history.

Besides the live-in access checks and statement transformation performed during speculative execution, a mechanism also needs to be developed to identify the live-in variables that are accessed during speculative execution. This is important because the values of live-in variables are the ones that require validation and copy-out operations after parallel threads finish their executions. Therefore, a *live-in table* is maintained for each parallel thread to track the live-ins accessed by the thread. The table has 3 fields as shown below.

NonSpecAddr	SpecAddr	WriteFlag

When a live-in variable is first accessed by a parallel thread T, an entry for this variable is created in the table. The *NonSpecAddr* field and *SpecAddr* field contain the live-in variable's non-speculative address and speculative address respectively. The *WriteFlag* field shows whether the live-in has been ever updated during speculative execution. Once a write access to this variable is encountered, this field is set to true. Subspaces and the live-in table are maintained during the speculative execution; they are crucial for performing the misspeculation detection and copy-out operations which will be described in the next two subsections.

4.2 Handling Speculative Results

After a parallel thread finishes its task, the main thread needs to perform misspeculation check to validate the speculatively computed results. If the speculation succeeds, the results are copied back to the non-speculative state. Otherwise, recovery procedure is initiated so that the correct values can be recomputed. In our approach, the results computed by parallel threads are handled in the same order as the order in which the tasks assigned to them were created. The in-order result-handling ensures the updates to non-speculative state memory is consistent with the sequential program semantics.

A misspeculation occurs if a speculatively read live-in variable has been updated during an earlier speculative task. On the other hand, if a parallel thread updates a live-in variable, then all other parallel threads that are using this variable to perform later tasks should be considered as failed because they are using an incorrect value. Based on this idea, a scheme is developed for the main thread to perform the copy-out operations and misspeculation checks. This scheme is presented in Figure 11.

Copy-out. Each parallel thread has a flag *SpecFail* indicating if its speculation has failed. When a parallel thread finishes its speculative task and the main thread determines that it is this thread's turn to commit its results, this flag is examined. If the flag indicates that no misspeculation occurred (line 1), the main thread

```
T.LTable: live-in table of a parallel thread T;
T.SepcFail: the flag indicating a failed speculation of T;
T.FailedAt[]: the live-in variables which cause the
speculation of thread T to fail;

1:  if (T.SpecFail != True) {
2:    foreach entry e in T.Ltable {
3:      if (e.WriteFlag == True) {
4:        val = GetLatestValue(e.SpecAddr);
5:        CopyBack(e.NonSpecAddr, val);
6:        foreach thread t executing a later speculative task {
7:          if (e.NonSpecAddr is in the t.LTable ){
8:            t.SpecFail = True;
9:            add e.NonSpecAddr into t.FailedAt[] ;
10:         } //endif
11:       } //end foreach
12:     }//end if
13:   }//end for each
14  }
15: else { //recovery
16:   pc = GetRecoveryPC(T.FailedAt[]);
17:   sid = GetRecoverySpaceID(T.FailedAt[]);
18:   ask thread T to reexecute from instruction pc using
      the latest values of live-in variables, and
      the values stored in the latest subspace whose SID
      is no more than sid for none-live-in variables ;
    }
```

Figure 11. Misspeculation Checks And Recovery.

starts to copy back the results. Specifically, the main thread goes through this parallel thread's live-in table (line 2) and identifies the modified live-in variables (line 3). The *WriteFlag* of a live-in variable is set when an access to this variable is a write. For each modified live-in variable, the main thread finds its latest value (line 4) and copies the value to the non-speculative state (line 4). The non-speculative address of a live-in variable is available as it is stored in the live-in table when the live-in variable is first accessed.

Misspeculation Check. The misspeculation check is performed by executing lines 6 to 11. In particular, when a live-in variable is committed, the main thread searches the live-in table of all other threads that are executing later speculative tasks. If this variable is found in another thread *t*'s live-in table (line 7), then thread *t* is using a stale value because the value of this variable has just been updated. As a result, the *SpecFail* flag of *t* is set to *true* (line 8). This variable identified by its non-speculative address is also added into another thread attribute *FailedAt* (line 9). The variables stored in *FailedAt* are used to identify the starting point of reexecution during the misspeculation recovery. Note that the use of *WriteFlag* ensures that multiple writes to a live-in variable within one task will not cause later tasks to be recovered multiple times.

Recovery. If the *SpecFail* flag of a thread *t* is set to *True*, then reexecution is required for recovery. The *FailedAt* attribute of *t* indicates which live-in variables caused the speculation to fail. Since, when each live-in variable is first read, the instruction address of the read and the space ID before the read are recorded (Figure 10, line 2), the main thread can retrieve these two values of the first accessed live-in variable by calling another two auxiliary functions *GetRecoveryPC* and *getRecoverySpaceID* (lines 16-17). These two values determine the starting point of the reexecution. The main thread now asks thread *t* to reexecute from instruction *pc*. During the reexecution, the latest value of every live-in variable is used. For other variables, their values stored in the latest subspace whose SID is no greater than *sid* are used. This can be done by searching for the state history of each variable. In this way, the reexecution is consistent with the sequential semantics and all incorrect speculatively computed results are simply discarded. More importantly, all

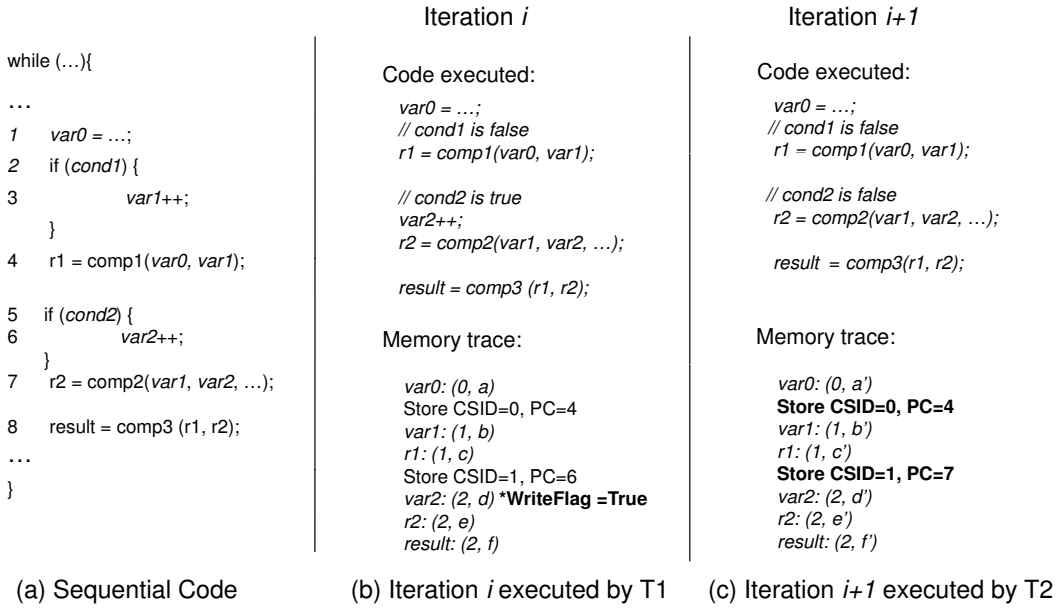

Figure 12. An Example Of Recovery.

correct speculatively computed results are reused because they are stored in different subspaces.

Figure 12 shows an example. Figure 12(a) shows the sequential code where two live-in variables *var1* and *var2* can be observed. Consider iteration *i* and *i+1*, which are speculatively executed by threads *T1* and *T2* in parallel. The executed code and memory operations for these two threads are shown in figures (b) and (c) separately. During *T1*'s execution, *T1* updates *var2*, and hence, the *WriteFlag* of this variable is set to true. When the main thread performs the misspeculation check, it searches for all other threads that are currently using *var2* and sets the corresponding flag *SpecFail* to true (Figure 11). In this example, *T2.SpecFail* is set to true and *var2* is added to *T2.FailedAt*. When the main thread examines *T2*'s results, it realizes that *T2*'s speculation has failed because of *var2*. Hence, it retrieves the CSID and PC of the recovery point that corresponds to *var2* (CSID=1 and PC=7 in this case), and sends the latest value of *var2* to *T2*. With such information, *T2* now can start the incremental recovery. In particular, it reexecutes the code from the instruction whose PC is 7, and uses the latest value of *var2* and the value stored in space 1 for all other variables.

Handling Exceptions. In certain situations, executing different loop iterations in parallel may cause the program to exhibit abnormal behavior. For instance, suppose an integer variable *var* is 0 initially. In the first iteration, it is updated to some integer that is larger than 0 and then used as a divisor in the second iteration. If these two iterations are speculatively executed in parallel, the thread executing the second iteration will throw a floating point exception, which will terminate the program. To solve this problem, all signals that lead to an abnormal termination are captured and handled by customized signal handlers. In these handlers, *SpecFail* flag of the corresponding parallel thread is set to true and recovery point is set to the beginning of the task. Therefore, during the recovery, the parallel thread can reexecute its task with the latest values of all live-in variables and exceptions will not take place any more. This exception handling mechanism guarantees that the parallelized application is strictly consistent with the sequential version.

5. Experiments

5.1 Experimental Setup

To study the effectiveness of our techniques we identified five benchmarks shown in Table 1 for which misspeculation rate is input dependent. The first three columns of the table give the name, lines of source code, number of live-in variables for these programs. Note that the outermost loops in these benchmarks can be speculatively parallelized. Exploiting such parallelism when certain inputs are used yields good speedups, especially for program `CRC32` [37]. Thus, `CRC32` is selected in addition to the four SPEC programs for comparison to previous work. For each benchmark in our experiments, three different inputs were used which give rise to cross-iteration dependences and thus misspeculations at a low ($< 1\%$), medium (around 40%), and high (around 80%) frequency as shown by the last three columns. Experiments were conducted using these different inputs to evaluate the effectiveness of incremental recovery and its comparison to *discard-all* recovery.

Name	LOC	# of Live-ins	Cross-iteration Dep. Frequency		
			low	medium	high
`197.parser`	9.7K	8	1%	49.3%	82.4%
`130.li`	7.8K	6	2%	40%	80%
`256.bzip2`	2.9K	9	0.5%	38.2%	79.2%
`255.vortex`	49.3K	11	0.5%	43.2%	81.3%
`CRC32`	0.2K	1	1%	40%	80%

Table 1. Benchmark Description.

In the experiments, the parallel version of every benchmark program is generated through a source-to-source transformation. Specifically, to support the space and thread decoupling scheme, every basic type in C (e.g., int, char etc.) is redefined as a new class in C++. The class contains not only the original type, but also the type of state history in parallel spaces. When a variable is declared to be of a class base type, its non-speculative storage is allocated during the compilation and its state history for each thread is created and dynamically maintained as shown in Figure 10. The parallel threads are created and controlled using a template similar to

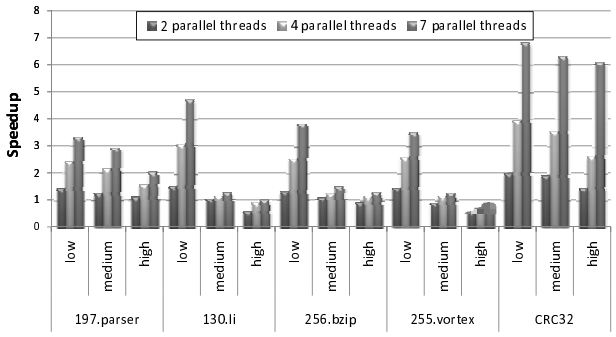

Figure 13. Speedups: Incr. Recovery vs. Sequential Execution.

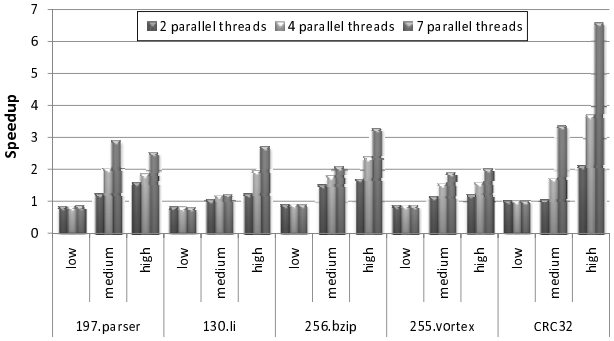

Figure 14. Speedups: Incremental Recovery vs. Discard-all.

one described in [37], but written in C++. The runtime system that detects the misspeculation and handles the speculatively computed result described in this paper was also implemented in C++.

All the experiments were conducted under CentOS 4 OS running on a *dual quad-core* Xeon machine with 16GB memory. Each core runs at 3.0 GHz. Since there are a total of 8 cores on this system, we measured the performance when 2, 4, and 7 speculative threads are created in addition to the main thread that coordinates the activities of the parallel threads.

5.2 Performance Results

5.2.1 Speedups over Sequential Execution

The first experiment conducted measured the speedups resulting from speculative parallelization that employs incremental recovery for different inputs and different number of parallel threads. Figure 13 shows the speedups when using 2, 4, and 7 speculative threads in comparison to executions of sequential versions of the applications. As we can see, for each benchmark, on any given input, the performance of the benchmark improves as the number of parallel threads spawned increases. When 7 threads are executed on `input-low`, all benchmarks obtained the best speedups which range from 3.3 to 6.8.

When `input-medium` and `input-high` are used, performance gains can still be observed in most cases, but these gains drop dramatically for `130.li`, `255.vortex` and `256.bzip`. Especially when running on `input-high`, the first two benchmarks experience slowdowns as misspeculations more than wipe out the benefits of parallelism. However, for `197.parser` and `CRC32`, speedups can be observed even when `input-high` is used. In these two programs, only a small number of instructions need to be reexecuted when a misspeculation occurs and hence the performance is not limited by the misspeculation rate. The highest speedups achieve on `input-medium` and `input-high` are 2.9 and 2.0 for `197.parser`, 6.3 and 6.1 for `CRC32` respectively.

5.2.2 Comparison With Discard-All Scheme

Speedup Comparison. Next we show the advantage of incremental recovery over discard-all. The speedups of incremental recovery based speculative parallelization over discard-all based speculative parallelization are shown in Figure 14. We observe that for all benchmarks running with `input-medium` and `input-high`, incremental recovery has much better performance than discard-all scheme. It brings more benefit when more threads are used. As shown in Figure 14, the speedups observed when 7 threads are used, range from 1.2x (`130.li`) to 3.3x (`CRC32`) for `input-medium` and from 2.0x (`255.vortex`) to 6.6x (`CRC32`) for `input-high`.

The reason is that the misspeculation rates become higher when more threads are used, which significantly slows down the parallelized version that uses the discard-all scheme. In contrast, when incremental recovery is applied, only part of the speculatively computed results need to be recomputed and the overhead of recovery is greatly reduced. As a result, incremental recovery can still speed up the sequential program in most cases as shown in Figure 13. It is worth noting that for `255.vortex` and `130.li` running with `input-high`, Figure 13 indicates that incremental recovery technique cannot speed up the sequential executions. However, the performance of the original parallelized version is even worse. That explains why the speedups are still obtained in this experiment.

In the case of `input-low`, the performance of using incremental recovery is worse than the original scheme. The performance loss is about 15% on an average. This is because of the overhead of the recovery system. In particular, every load or store has to be performed on a state history – a list of the pairs of space ID and value. This is more expensive than operating on a single value. The results of the experiment shows that incremental recovery is much more effective if the misspeculation rate is very high and the discard-all scheme is better otherwise.

Misspeculation Comparison. Another reason for incremental recovery technique performing better on `input-medium` and `input-high` is that it reduces the number of misspeculations. In our scheme, copy-on-write scheme is applied (see Figure 10), and using incremental recovery technique can increase the chance of a later iteration reading the correct values of the variables that are modified in an earlier iteration as discussed in section 3. The reason is that our technique allows the recovery to be performed faster and thus it speeds up the result-committing stage of every thread. Experiments were conducted to measure this effect. In particular, the misspeculation rate is collected for both incremental recovery technique and the discard-all technique. The misspeculation rate reduction is then computed based on these two numbers and the results are shown in Figure 15. Note that the comparisons are only made on `input-medium` and `input-large`.

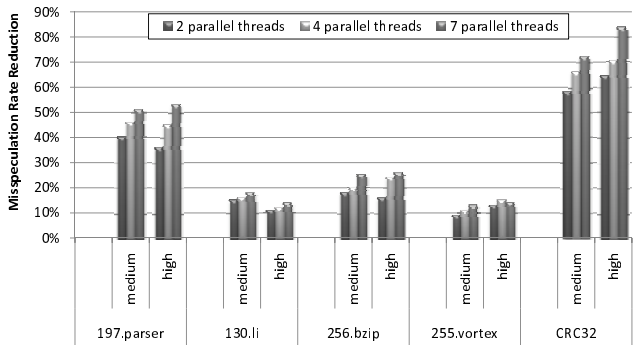

Figure 15. Misspeculation Rate Reductions.

From the figure, one can see that for `197.parser` and `CRC32`, up to 50% and 85% misspeculations are respectively eliminated. As mentioned earlier, in these two programs, only one statement, which defines the live-in variable, needs to be recomputed during the recovery. With incremental recovery technique, the live-in variable can be committed faster, which increases the chance of later threads getting the correct version. As a result, performance of these two benchmarks is significantly improved by our technique (see Figure 13). For other programs, the number of reexecuted statements during the recovery is far greater, and hence the observed misspeculation reduction is around 20% for `256.bzip`, 10% for `130.li`, and 10% `255.vortex` on an average.

Recovery Time Comparison. To further understand the difference between the speculative execution with and without incremental recovery technique, we conducted another experiment to measure the fraction of time that is spent in recovery mode (i.e., reexecution of previously failed tasks) during the speculative execution. We used seven parallel threads in this experiment and the fraction numbers are averaged across all parallel threads. Figure 16 shows the results.

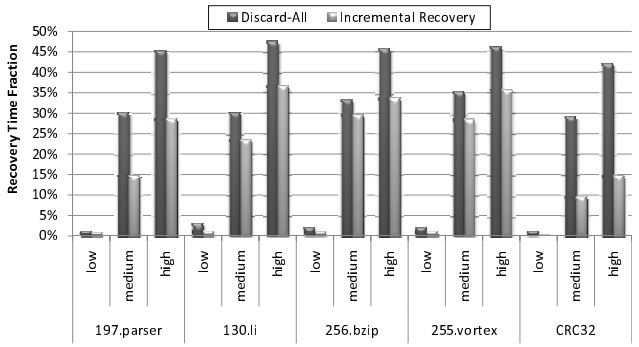

Figure 16. Recovery Time Comparison.

From the figure, we can see that for `input-low`, less than 2% of the execution time of each parallel thread is spent on recovery during both executions. Thus, incremental recovery does not provide any benefit. However, when `input-medium` is used, difference can be noticed. With incremental recovery, the fraction of time spent in recovery ranges from about 10%(`CRC32`) to 28% (`255.voertex`). With discard-all scheme, this number is around 30% for all benchmarks. This implies that very high misspeculation rates are encountered during the parallel executions of these benchmarks and about one third of the time is spent on reexecutions. For `input-high`, recovery takes about 45% of the execution time of each thread when the original discard-all scheme is used in the parallelization. With incremental recovery, however, each thread of parallelized `197.parser` and `CRC32` spent less than 30% of time on recovery. In other programs, these numbers are around 35%, which are still smaller than discard-all scheme.

Finally, recall that benefit of incremental recovery over discard-all is due to two reasons – reduction in computation and reduction in misspeculation rate. We breakdown the contributions of these two factors in Figure 17. As we can see, although majority of the savings result from computation reduction, when misspeculation rates are high, significant savings result from reduction in misspeculation rate.

5.3 Overhead Analysis

Execution Time Breakdown. The execution time spent by each parallel thread can be broken down into three different categories: *communication*, *recovery*, and *speculative computation*. Figure 18 shows the results. They are measured and averaged across all seven

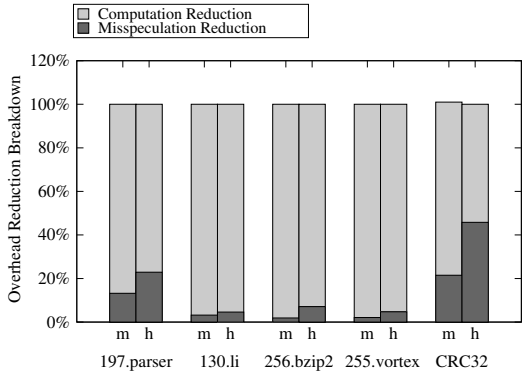

Figure 17. Breakdown: Reuse and Misspeculation Reduction.

parallel threads. As we can see, the time spent on the communication is only a small portion for all programs regardless of the input. This implies that all parallel threads are busy performing computations. For all parallelized programs running on `input-low`, each thread spends most of the time on speculative execution. When the input changes, more misspeculations take place, and thus more time is spent on the recovery. In the case of `256.bzip2`, `130.li` and 255.vortex running on `input-high`, about 35% of the time is spent on reexecuting the speculative computation.

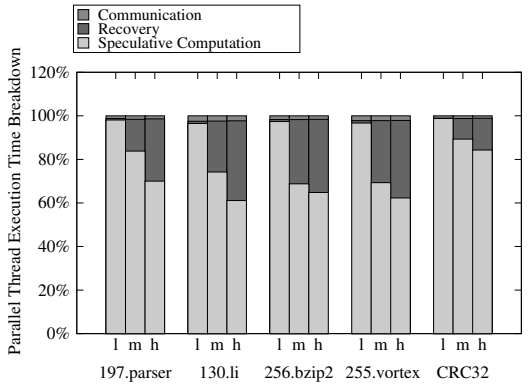

Figure 18. Time Breakdown: Speculative Threads.

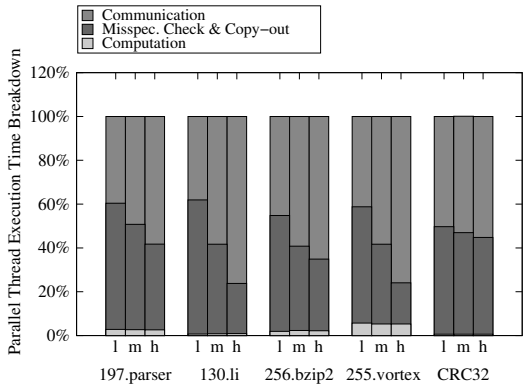

Figure 19. Time Breakdown: Main Thread.

Figure 19 shows the time breakdown for the main thread. Three categories are considered: *communication*, *misspeculation checks & copy-out*, and *computation*. Different from parallel threads, the main thread spent a large portion of the execution time on communication. Another significant portion is spent on misspeculation

checks and committing results. When different inputs are used to create more misspeculations, one can observe that communication time increases, because parallel threads fail more often. As a result, the main thread has to spend more time on waiting for the correct results to be produced.

Space Overhead. Maintaining multiple spaces for one thread consumes more space. An experiment is conducted to monitor the peak value of memory consumption. The memory used by the corresponding sequential program is considered as the baseline. Figure 20 shows the space overhead. As expected, using more threads consumes more space because each variable accessed by a parallel thread takes more memory space in this technique. When 7 parallel threads are used, the largest overhead is observed which ranges from 3.2 to 5.1 across all benchmarks. Note that the data in this figure is for `input-high` only because the space overhead is not sensitive to the misspeculation rate.

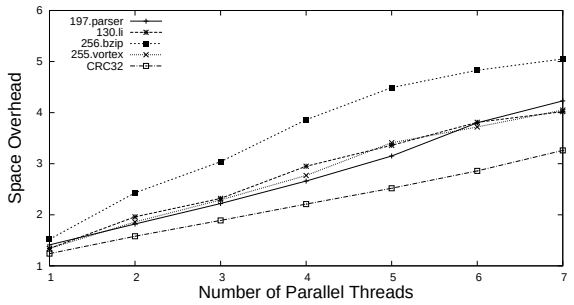

Figure 20. Space Overhead.

6. Related Work

A great deal of research has been carried out on hardware TLS [6, 10, 14, 19, 31, 33]. While the compiler is responsible for speculatively parallelizing sequential code, hardware is provided to handle the difficult tasks of detecting and handling misspeculations. Specialized hardware structures are used, such as special buffers [10, 23], versioning cache [8], or versioning memory [7]. Unfortunately, none of these features are available in commercial multicore processors of today. Therefore an attractive alternative avenue of optimistically extracting parallelism from programs is based upon a purely software realization of TLS.

Most of the software TLS systems use state separation to support speculations [2, 3, 5, 9, 13, 26–28, 35–38]. To identify speculatively parallelizable loops, some of these works rely on static analysis [2, 3, 9, 26], some require the complete trace of a program to perform the parallelization [27, 28], and others use profiling technique to gather certain types of runtime information. As already described, Ding *et al*. proposed using processes to achieve state separation [5, 13] while in our prior work on CorD we proposed thread based state separation [37].

In recent work we proposed the use of multiple value predictions to reduce high misspeculation rate [35]. The idea is to predict the values of live-in variables that frequently cause misspeculations. However, its applicability is determined by the prediction accuracy. Moreover, due to the creation of multiple speculative versions for the same loop iteration, this work requires much more processor resources – one core per version. Thus, it is even more wasteful than discard-all as it always discards the results of all versions except the correct one. Our work, on the other hand, reduces the cost of misspeculation recovery and thus is far less expensive.

Instead of state separation, Kulkarni *et al*. proposed a rollback based speculative parallelization technique [15–18, 21]. They introduce two special constructs that users can employ to identify speculative parallelism. When speculation fails, user supplied code is executed to perform rollback. In other words, users must define the reverse computation in their programs. While the above work requires help from the programmer, our approach accomplishes the same tasks automatically.

Software pipelining is another parallelization technique that has been greatly studied [1, 12, 25, 34, 39]. Different from DO-ALL parallelization, these techniques partition each loop iteration into several pipeline stages and each stage is executed on a different core. While some of them focus on multicore processors [25, 39], others target stream and graphic processors [1, 12, 34]. Hardware versioning memory is used to separate non-speculative and speculative results.

Transactional memory (TM) [4, 11, 22] has been an active area of research. It is designed to enforce the atomicity of shared memory accesses in parallel programs and cannot be directly used to parallelize sequential programs due to several reasons [20]. First, STM needs special mechanisms to avoid or resolve dead-lock and live-lock situations. Second, STM aims at achieving good throughput and fairness. This requires STM to consider the priorities of transactions [32]. Besides, STM internally uses locks to prevent data races [33] and barriers to ensure strong atomicity [29, 30] and in-order commit [20]. This results in high runtime overhead for STM while providing a convenience for programmers writing parallel applications. Mehrara *et al*. proposed customized STM for speculative parallelization [20]. Their work assumes dependent variables can be identified at compile time and requires a set of special registers to track such variables. Raman *et al*. also proposed the use of multi-threaded transactions to support speculative execution [24]. However, their work focuses on applying transactions on software pipelining parallelization instead of DO-ALL parallelization.

7. Conclusion

In this paper we presented an *incremental recovery* technique to reduce the overhead of misspeculation recovery in an existing implementations of SS-TLS. The key idea was to decouple the creation of each parallel thread from the creation of speculative memory spaces. Use of multiple spaces during speculative execution allows the speculatively computed results to be stored and reused when a misspeculation occurs. Our experiments show that for a set of programs with inputs causing around 40% and 80% misspeculation rates, applying incremental recovery technique can achieve 1.2x-3.3x and 2.0x-6.6x speedups respectively in comparison to the discard-all recovery scheme used in prior works.

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

References

[1] I. Buck. *Stream computing on graphics hardware*. PhD thesis, Stanford, CA, USA, 2005.

[2] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, 2003.

[3] M. H. Cintra and D. R. L. Ferraris. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):562–576, 2005.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII*, pages 336–346, 2006.

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

of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pages 223–234, 2007.

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

[7] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *Transactions on Architecture and Code Optimization*, 2(3):247–279, 2005.

[8] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, 1998.

[9] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–12, Washington, DC, USA, 1998. IEEE Computer Society.

[10] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, 1998.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[12] 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.

[13] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 157–168, 2009.

[14] V. Krishnan and J. Torrellas. The need for fast communication in hardware-based speculative chip multiprocessors. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 24–33, 1999.

[15] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, 2009.

[16] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA*, pages 217–228, 2008.

[17] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS XIII*, pages 233–243, 2008.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[19] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 365–372, 1999.

[20] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 166–176, 2009.

[21] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, 2010.

[22] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII*, pages 359–370, 2006.

[23] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 204–215, 2001.

[24] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 65–76, New York, NY, USA, 2010. ACM.

[25] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, pages 114–123, 2008.

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

[27] P. Rundberg and P. Stenstrom. Low-cost thread-level data dependence speculation on multiprocessors. In *In Fourth Workshop on Multithreaded Execution,Architecture and Compilation*, 2000.

[28] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *J. Instruction-Level Parallelism*, 3, 2001.

[29] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 181–194, 2008.

[30] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, 2007.

[31] G. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, 1995.

[32] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009.

[33] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00: In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, 2000.

[34] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[35] C. Tian, M. Feng, and R. Gupta. Speculative parallelization using state separation and multiple value prediction. In *ISMM '10: Proceedings of the 2010 International Symposium on Memory Management*, 2010.

[36] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 63–73, 2010.

[37] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, 2008.

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

[39] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 2007 International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, 2007.