Supporting Speculative Parallelization in the Presence of Dynamic Data Structures

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

Abstract

The availability of multicore processors has led to significant interest in compiler techniques for speculative parallelization of sequential programs. Isolation of speculative state from non-speculative state forms the basis of such speculative techniques as this separation enables recovery from misspeculations. In our prior work on CorD [35, 36] we showed that for array and scalar variable based programs copying of data between speculative and non-speculative memory can be highly optimized to support state separation that yields significant speedups on multicore machines available today. However, we observe that in context of heap-intensive programs that operate on linked dynamic data structures, state separation based speculative parallelization poses many challenges. The copying of data structures from non-speculative to speculative state (copy-in operation) can be very expensive due to the large sizes of dynamic data structures. The copying of updated data structures from speculative state to non-speculative state (copy-out operation) is made complex due to the changes in the shape and size of the dynamic data structure made by the speculative computation. In addition, we must contend with the need to translate pointers internal to dynamic data structures between their non-speculative and speculative memory addresses. In this paper we develop an augmented design for the representation of dynamic data structures such that all of the above operations can be performed efficiently. Our experiments demonstrate significant speedups on **a real machine** for a set of programs that make extensive use of heap based dynamic data structures.

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

General Terms Performance, Languages, Design, Experimentation

Keywords Speculative Parallelization, Multicore Processors

1. Introduction

The thread level speculation (TLS) [7, 9, 10, 16, 21, 25, 31, 33, 37] technique has become increasingly important due to the availability of multicores. It allows the compiler to optimistically extract parallelism from sequential programs. In particular, the compiler creates multiple threads to execute different portions of a program in parallel optimistically assuming that no dependences exist between these selected portions. TLS detects misspeculations, i.e. violations of these assumptions by detecting if dependences that were assumed to be absent manifest at runtime. To ensure the correctness of the execution, TLS must appropriately handle the results of speculative computations. Although considerable research work has been carried out on TLS, most of the work is hardware based and not ready for use. This is due to the architectural redesigns [7, 21, 31, 37] requiring non-trivial hardware changes (e.g., special buffers [10, 25, 26], versioning cache [9], versioning memory [8]) for detecting misspeculations and handling speculative results which have not been incorporated in commercial multicores.

Another avenue of optimistically extracting parallelism from programs is based upon a purely software realization of TLS. While developing an efficient software implementation of TLS is challenging, the benefits of this approach are clear as it can be applied to existing widely available multicore systems. Recently, software based TLS techniques have been proposed in [5, 17–19, 34–36]. These techniques have been shown to be quite effective in optimistic parallelization of streaming applications on multicores. While the work in [17–19] requires the programmer to provide recovery code, the works in [5, 14, 34–36] are based upon realization of state separation with no programmer help. In [5, 14] Ding et al. achieve state separation by creating separate processes for a non-speculative and speculative computations – since each process has its own address space, state separation is achieved. In CorD [34–36] we use a single process with multiple threads - a nonspeculative main thread and speculative parallel threads. To achieve state separation, storage is allocated separately for the threads and copying operations are performed to transfer data between threads.

While the above software TLS techniques have been shown to achieve speedups without requiring any modifications to the architecture of existing multicores, they cannot be used for programs with heap based dynamic data structures. In case of Ding et al. [5], shared variables which are assumed not to be involved in dependences must be allocated on separate pages. This is because page level access tracking is employed to detect misspeculations. However, in programs that employ dynamic data structures that may contain millions of data items, it is not practical to allocate each of them on a separate page. Thus, the process based approach proposed by Ding et al. [5] is not suitable for dynamic data structures. The work proposed by Kulkarni et al. [17–19] handles dynamic data structures but is not general as it is aimed at work list based applications.

In our work on CorD [34–36], shared variables that may be modified by speculative threads are copied into speculative threads memory space and modifications to these variables are tracked to detect misspeculations. In case of dynamic data structures inability to separate parts of a large dynamic data structure that are shared

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

from those that are not shared will lead to too much copying and access tracking overhead as safe assumption is consider the entire data structure as shared. Thus, additional mechanisms are required to prevent unnecessary copying and checking in order to make our approach practical for programs with large dynamic data structures.

In this paper we develop mechanisms that enable CorD to efficiently supports speculative execution of programs that operate on heap based linked *dynamic* data structures. In particular, we address the following challenges in the context of heap-intensive programs:

- *What to Copy-In ?* Complexities of pointer analysis makes it difficult to identify the portion of the dynamic data structure that is referenced by the speculative computation. Conservatively copying the entire data structure may not be practical when the size of the data structure is very large.
- *How to Copy-Out ?* The copying of updated data structure from speculative state to non-speculative state is made complex due to the changes in the shape and size of the dynamic data structure that may be made by the speculative computation.
- *How to handle internal pointers ?* In addition, both copy-in and copy-out operations must contend with the need to translate pointers internal to dynamic data structures between their non-speculative and corresponding speculative memory addresses.

In this paper we address all of the challenges outlined above. First, we propose the *copy-on-write* scheme which limits the copying to only those nodes in the dynamic data structure that are modified by the speculative computation. When a speculative thread *writes* to a node in a dynamic data structure for the first time, the node is copied into speculative state. If a speculative thread *only reads* a node in the non-speculative state, it is allowed to directly access the node from the non-speculative state. Second, we present the *heap prefix* augmentation for the representation of dynamic data structures and *double pointers* representation for internal pointers. These representations enable the translation of addresses during copy-in and copy-out to be efficiently handled, the implementation of runtime *access checks* preceding data structure accesses to be optimized, and runtime *misspeculation checks* preceding copy-out operations to be optimized. Our experiments demonstrate significant speedups on **a real machine** for a set of programs that make extensive use of heap based dynamic data structures.

2. State Separation for Dynamic Data Structures

We begin by briefly summarizing our CorD [35] execution model and then discuss the challenges that must be overcome to speculatively parallelize programs with dynamic data structures.

2.1 State Separation In CorD

(The CorD Model) As shown in Fig. 1, separate memory is allocated to hold the non-speculative state of the computation and the state corresponding to the speculative threads. When a speculative thread is created, it speculatively *copies-in* values of needed variables from non-speculative memory (also called D space) to speculative memory (also called P space), performs the speculative computation, and if speculation is successful, the results produced are *copied-out* from speculative to non-speculative state. To enable copying-out of values, a *mapping table* is used by each speculative thread. In the mapping table, each copied-in variable has an entry, which have five fields, namely *D_Address*, *P_Address*, *Length*, *WriteFlag* and *Version*. The first three fields define the address mapping information. When a variable is modified in P space, its *WriteFlag* is set which indicates it needs to be copied-out if the speculation succeeds. If misspeculation occurs, i.e. the speculative computation does not conform to the sequential program semantics, the speculative state is discarded and computation is repeated.

To implement misspeculation detection, version numbers are maintained for each variable. In particular, when a thread performs a copy-in operation for a variable, it copies the current version number of the variable into the *Version* field in the variable's mapping entry. The current version number is maintained by the main thread for each variable, and it is incremented each time the variable is copied out. When detecting the misspeculation, the main thread compares this version with the one stored in the *Version* field for each variable. If the versions of every speculatively-read (copied-in) variable are the same, then the speculation is successful. This is because these variables used by a speculative thread have not changed from the time they were read until the time at which speculative results are produced. However, if the version of any variable has been changed by an earlier speculative thread being executed in parallel on another core, then misspeculation occurs. The transfer of data from (to) D space to (from) P space at the start (end) of a speculative computation must be highly optimized to minimize the impact of copying overhead on execution time performance. In [35, 36] we developed techniques for achieving such optimizations for programs that mainly operate upon data held in global arrays and scalar variables.

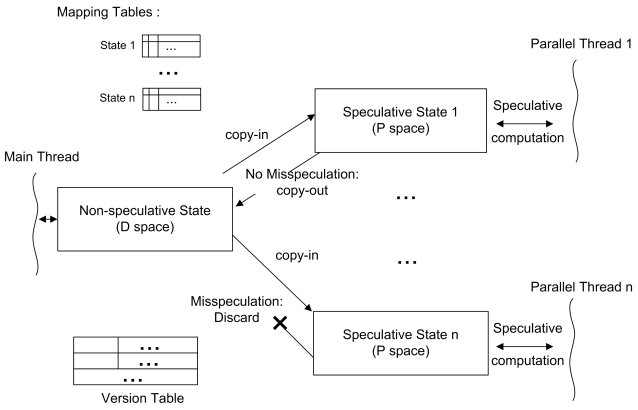

Figure 1. Separating Speculative And Non-speculative State.

(An Example) Fig. 2 illustrates CorD using an example. Fig. 2(a) shows a sequential *while* loop which basically perform some computation using variable *key*. If the computation returns FALSE, the value of *key* is updated. Fig. 2(b)-(d) shows a possible situation where two consecutive iterations are executed in parallel under the CorD model. As shown in Fig. 2(d), the non-speculative address of variable *key* is *0xA* and its current version *v* is stored in the version table.

From the code executed by the main thread, one can see that two parallel threads are created. Before sending the start signal to each parallel thread, the main thread creates two local copies of *key* – *key'* and *key"* for thread 1 and thread 2 respectively. It also adds a mapping entry into the mapping table of each parallel thread. The entry shows the D address, P address, length and current version of *key*. The *WriteFlag* is also set to false.

When a parallel thread executes code speculatively, it uses its own local copy of *key* as shown in Fig. 2(b) and 2(c). Since the computation in thread 1 returns FALSE, thread 1 must update *key'* and set the *WriteFlag* to true. Thread 2 simply performs the computation using *key"* and does not need to change the *WriteFlag* in the mapping table. Clearly, thread 2 is using a stale value of *key* because the previous iteration has changed the value. This misspeculation is detected by the main thread as shown in Fig. 2(d). When thread 1 finishes, the main thread compares the version of *key* stored in the mapping table with the one stored in the version table. Since they are the same, copy-out operations are performed and

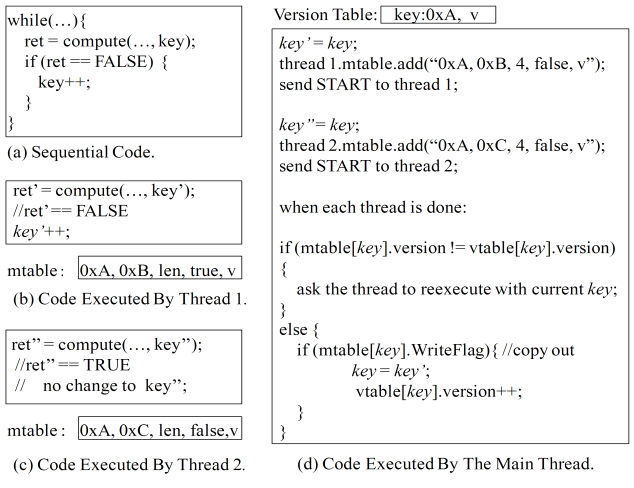

Figure 2. Separating Speculative And Non-speculative State.

the version of *key* is incremented. When thread 2 finishes, the main thread finds a mismatch between two versions of *key* (i.e., *v* and *v+1*), and thus asks this parallel thread to re-execute the code using the latest value *key*. Note that the version comparison statement shown in Fig. 2(d) should be performed for all copied variables and a misspeculation is reported if any mismatch is found.

From the above example one can see how state separation is achieved using copying. Together with the version comparison based misspeculation detection, CorD is able to aggressively and safely exploit the parallelism in a sequential loop.

2.2 Challenges For Dynamic Data Structures

A dynamic data structure consists of large number of nodes such that each node contains some data fields and pointer fields. The pointer fields are used to link together the nodes in the data structure (e.g., link lists, trees, queues etc.). Such data structures are also called dynamic data structures because the shape and size of the data structure can change as the program executes. Size of the data structure changes as nodes are added or removed and changes in link pointers can further change the shape of the data structure. The memory for each node is dynamically allocated from the heap when the node is created and freed by returning it to the heap when the node is eliminated. Dynamic data structures are extensively used in wide range of applications. The applications used in our experimentation are C programs with following characteristics:

- each node is allocated and deallocated through explicit calls to library functions *malloc* and *free*; and
- nodes are accessed through pointers variables that point to the nodes such as pointer fields that link the nodes in the data structure.

The state separation based speculative parallelization for programs using dynamic data structures is much more challenging than for those using scalar variables or static data structures such as arrays. In this section, we will describe the challenges and develop techniques to address them.

Given a parallelized computation which consists of a non-speculative main thread that spawns multiple speculative threads, in CorD model state separation is achieved by performing speculative computations in separate memory space. While for array or scalar variables the separation can be simply achieved by creating a copy of such variables in the speculative thread, achieving state separation for programs using dynamic data structures poses many challenges. A dynamic data structure may contain millions of nodes (e.g., the program *Hash* in our experiments creates 3 million nodes at runtime). This leads to a large overhead due to copying operations, mapping table accesses, and misspeculation checks. Moreover, need for address translation arises because a node may have many pointers pointing to it and after the node has been copied, accesses via these pointers must be handled correctly.

```
typedef struct node{
        int key;
        int val;
        struct node *next;
}
NODE *head;

while (...) {
        ...
1:      NODE *tmp = find_key(head, key);
2:      if (tmp!=NULL and tmp!=head) {
3:          NODE *prev = get_prev_node(head, tmp);
4:          prev → next = tmp → next;
5:          tmp → next = head;
6:          head = tmp;
7:      }
8:      else {
9:          if (!tmp) { //update the lru queue
10:             //insert the new node
11:             NODE *n = (NODE *)malloc(sizeof(NODE));
12:             n→ key = ...;
13:             n→ val = ...;
14:             n→ next = head;
15:             head = n;

                //delete the least-recent used node
16:             NODE *m = get_second_last_node(head);
17:             free(m→ next);
18:             m→ next = NULL;
19:         }
20:     }
21:     if (writeflag)
22:     {
23:             head→val = ...;//modify data
24:     }
25:     ... = head→val;
26:     ...
}
```

Figure 3. Least Recent Use Buffer.

For a program using dynamic data structure, four types of changes to the dynamic data structure can be encountered:

- pointer fields in some nodes are modified causing the shape of the data structure to change;
- a new node is created and linked to the dynamic data structure;
- an existing node is deleted from the dynamic data structure causing the size of the data structure to change; and
- values of data fields in some nodes are modified.

Fig. 3 shows an example where all the above changes are encountered. In this example, a link list is used to implement a least-recent-use (LRU) buffer. A LRU buffer has a fixed length and it buffers most recently used data. When data is requested, we first search for any matching elements in LRU (line 1). If a match is found and the element is not in the front of LRU buffer, we move this element to the front by adjusting the pointers (lines 2-7). If no match is found, we create a new node, insert it in the front and delete the last node (lines 8-20). After the requested data is put at the front, we check if we need to modify the data (lines 21-24). Finally we read the data in this buffer (line 25). Let us assume that the requested data is frequently at the front of LRU buffer and branches

at lines 2, 8 and 21 are rarely taken. Thus, iterations in the *while* loop (lines 1-25) can be speculatively executed in parallel.

When a parallel thread is created, all pointers (*head*, *tmp*, *prev*, *m* and *n*) will have their own local storage in the corresponding speculative space. Note that *head*'s local storage (denoted by *head'*) will contain the content of *head*, as it is defined outside the parallelizable region. We use **P* to denote a node pointed to by pointer P. Next we describe the challenges via this example.

Overhead Challenge. As we can see the function call *find_key* at line 1 traverses all nodes in the LRU buffer. If the original CorD model is used, and the LRU buffer contains a million nodes at runtime, then the overhead of this traversal will be prohibitively high. First, the copying overhead is large as all these nodes can be potentially modified by speculative thread and hence will be copied into speculative state. Second, the mapping table that maintains correspondence between addresses in non-speculative and speculative memory is large, because for every copied node, we need to maintain a mapping entry. A large mapping table will lead to an expensive lookup and update. Last but not least, the version table is large. In the original CorD model, a global version number of each node is stored in the version table and used in performing misspeculation checks. In particular, if a node is modified, its global version is compared with its local version stored in the mapping entry. Searching for the global version number of a node in a large version table requires time. Besides, the search has to be done for all modified nodes. Finally, finding modified nodes from a large mapping table is very time-consuming. Consequently, the misspeculation check will be very time consuming.

Address Translation Challenge. A node in a dynamic data structure is allocated on the heap at runtime. Its address is stored in one or several pointer variables and its access is performed through such pointers. This creates the address translation problem. In particular, when a node is copied into speculative state by a copy-in operation, all pointers in speculative thread holding its address and being used in the computation must change their content to the address of the copied node accordingly. For example, in Fig. 3 line 5, a parallel thread will use *head'* which is holding a non-speculative state address as it is a copy of *head*. If the node **head* has been copied during the execution of function *find_key*, we must change the value of pointer *head'* to the address of the head node's local copy, and then assign this new value to *tmp* → *next*. This requires comparison of each pointer being accessed (in this example *head'*) with the addresses stored in the mapping table. Similarly, when a node is copied back to non-speculative state by a copy-out operation, all pointers containing its current local copy address need to hold its non-speculative state address now. In Fig. 3, when line 6 is executed, *head'* will point to the node **tmp*, a local copy of some node in the non-speculative state. Therefore, when the value of *head'* is copied back to *head*, the address needs to be changed to the address of that node. This can be done by consulting the mapping table with the address stored in *head'*.

If the branch at line 8 is taken, copying out node **n* is a challenge. At line 14, the *next* field of node **n* is assigned with the address of head node's local copy. However, when committing the result, the node **n* is represented as the starting address and length. Therefore, we cannot find which part of **n* is the starting address of the *next* pointer field, and thus, cannot translate the address. One solution might be to store the address of *next* pointer in the mapping table, but again, this may lead to an even larger mapping table as one node may contain multiple pointer fields that are modified. Similarly, when line 17 is executed, we also need the address translation so that the correct node in non-speculative state is deallocated. In the case of data field modification (line 23), however, there is no need for address translation.

2.3 Copy-On-Write Scheme

To address the overhead challenge, primarily we must find a way to reduce the number of nodes that are copied to speculative state. Therefore, we propose the use of *copy-on-write* scheme to limit the copying to only those that are modified by the speculative thread. A node in non-speculative state is allowed to be read by speculative threads. It will be copied into a thread's speculative state only when it is about to be modified. The copying is implemented through an *access check* – a block of code inserted by compiler to guard every node reference via a pointer. Based on the type of reference, read or write, *access check* code differs.

(Write Access) Upon a write to a node, the *access check* will determine if the node is already in the speculative state. If this is the case, the execution can proceed. Otherwise, the *access check* concludes that the address belongs to a node in non-speculative state. In this case the speculative thread must determine if this is the first write to the node and thus the node must be copied into the speculative space. However, if this is not the first write to the node, then the node has already been copied into speculative state. Thus, the address being referenced in the non-speculative state has to be translated into the corresponding address in the speculative state. This translation is enabled by ensuring that the mapping table is updated by creating entries for copied nodes. In other words, the access checks will consult the mapping table to determine if the current pointer refers to a node in speculative state or non-speculative state.

(Read Access) Upon a read to a node, the *access check* allows the execution to continue if the node is in speculative state. However, if the node is in non-speculative state, the *access check* stores the thread task ID for this node indicating when the node has been read. After this step, the execution can proceed. The thread task ID is an integer maintained by each thread. It is initially zero and incremented by one every time the thread is assigned a task to perform by the main thread. As we will see shortly, this information is used during misspeculation checks. It is worth noting that in this copy-on-write scheme, if a node is only read by a speculative thread, it will never have an entry in the mapping table. In other words, all mapping entries contain nodes that have been modified by the speculative thread.

(Example) Applying this scheme to the example in Fig. 3, we can observe the following two advantages. First, there is no need to copy every node in a dynamic data structure into speculative state when function *find_key* is executed. Therefore, the size of the mapping table is reduced. Second, the need for address translation is greatly reduced. In particular, if node **head* is never updated during execution, then we will not make a copy of this node. Consequently, the pointer *tmp→next* at line 5 and *n→next* at line 14 will get the correct non-speculative address of this node without address translation. For line 17, we can also simply mark the address stored in *m→next* as deallocated, instead of making a copy of a node **(m→next)* and then translating the address.

2.4 Heap Prefix

Although the copy-on-write scheme can reduce the size of mapping table, the access and update of this table may still impose large overhead on the parallel execution. In particular, for each heap access, the *access check* needs to consult the mapping table to see if a node has been copied or not. This requires a walk through the entire table. Similarly, the *misspeculation check* needs to search the version number of each modified node in the version table. This requires traversing the table multiple times. To efficiently perform the access checks and misspeculation checks we associate meta-data with each node that tracks certain information related to accesses of the node. We call this meta-data by the name of *heap prefix*. Next we describe the details of heap-prefix and show how

it is effective in reducing the overhead of using mapping table and version table.

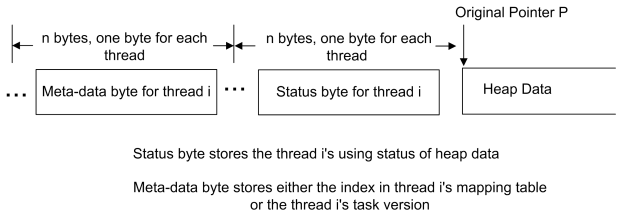

Figure 4. Heap Prefix Format.

For each memory chunk allocated on the heap, we allocate $2 * n$ additional bytes in front of it where n is the total number of speculative threads. These bytes represent the **heap prefix** which is used to store important information to assist in access checks. The format of the heap prefix is shown in Fig. 4. The first n bytes immediately before the program's original heap data are the *status bytes*. The additional n bytes are *meta-data bytes*. In the status byte, byte i represents the status for speculative thread i and it can represent four different possible status values. Status NOT_COPIED means the heap data has not been copied into thread i's speculative state. Status ALREADY_COPIED means the heap data has been copied into thread i's speculative space, and the index of this entry in thread i's mapping table is stored in the corresponding meta-data byte. Status ALREADY_READ means the heap data has been read by thread i, and the corresponding meta-data bytes stores the *task ID* of thread i. Status INTERNAL indicates that the node is already in the speculative state. Therefore, status NOT_COPIED, ALREADY_COPIED and ALREADY_READ only appear in heap elements of non-speculative state and status INTERNAL only appears in heap elements in speculative state. In the meta-data bytes, meta-data byte *i* stores either a index number of the mapping table of thread *i*, or the *task ID* of thread *i*.

Note that one can put the meta-data associated with each node in a different place and use hash function to locate it [32]. However, we tried the *hash-based* solution and observed that it caused over 6x slowdowns for the benchmarks we used. The reason is that a hash based lookup requires the execution of a hash function, which takes more time than performing a simple offset calculation. Thus, large number of lookups make the hash based solution yield visible slowdowns.

2.4.1 Implementing Access Checks

With the status bytes and meta-data bytes in the heap prefix, the *access check* for a heap node access in thread i can be implemented as shown in Fig. 5. Thread i's status s in the node's prefix is examined and following actions are taken.

If s is NOT_COPIED and the access is a read, s is updated to ALREADY_READ and the task ID of thread i is stored in the meta-data byte i (lines 7-10). If the access is a write, s is updated to ALREADY_COPIED. A new local copy of the node is then created with the corresponding status byte to be set to INTERNAL. Next, a mapping entry is added into the mapping table to reflect this copy operation and the index of the entry is stored in the meta-data byte i. Finally, the pointer points to the newly created node (lines 11-18).

If s is ALREADY_COPIED, then that means the node has been copied; thus, address translation is needed. Fortunately, the mapping entry can be quickly located through meta-data byte i and we only need to adjust the pointer to point to the address of the local copy (lines 20-23). Finally, if s is ALREADY_COPIED and the access is a write, then we perform the copy-in operation as when s is NOT_COPIED (lines 25-32). Otherwise, the access is a read or s is INTERNAL. In both cases, no further actions are required.

Fig. 6 shows an example of using heap prefix to perform access checks. First, assuming the node is allocated at 0xA in non-

```
1:  type = access type, READ or WRITE;
2:  p  = the pointer holding the starting address
          of the node being accessed;
3:  len  = the size of the node being accessed;
4:  s = thread i's status at *p;
5:  m = thread i's meta-data byte at *p;

6:  if (s == NOT_COPIED)
7:    if (type == READ) {
8:        s = ALREAD_READ;
9:        m = task_ID;
10:   }
11:   else { // type == WRITE
12:       s = ALREADY_COPIED;
13:       pointer q = make_copy(*p);
14:       set thread i's status at *q to INTERNAL;
15:       index = update_mapping_table(p, q, len);
16:       set thread i's meta-data byte at node to index;
17:       p = q;
18:   }
19: }
20: else if (s == ALREADY_COPIED) {
21:   index = thread i's meta-data byte at node;
22:   p = get P address from mapping table entry index;
23: }
24: else if (s == ALREADY_READ) {
25:   if (type == WRITE) {
26:       s = ALREADY_COPIED;
27:       pointer q = make_copy(*p);
28:       set thread i's status at *q to INTERNAL;
29:       index = update_mapping_table(p, q, len);
30:       set thread i's meta-data byte at node to index;
31:       p = q;
32:   }
33: }
34: else  { //s == INTERNAL
35:   ; //do nothing
36: }
```

Figure 5. Access Checks.

speculative state (D space), consider the execution of speculative thread T3. Before any reference to this node in T3, the status byte for T3 shows that the node has not been copied into its speculative state (P space) yet (as shown on the left).

Suppose there is a write to this node during the execution, the access check will make a copy of this node as it sees the status is NOT_COPIED. Therefore, the following actions will be taken. A new node is allocated at 0xB in P space and initialized with the original node value; a mapping entry is created in the mapping table (its index is x); T3's status of the original node is changed to ALREADY_COPIED indicating that this node has been copied into speculative memory, and the corresponding meta-data byte of T3 stores the index of mapping entry (x); T3's status of the copied node is set to ALREADY_COPIED which means this node is already in speculative state. In the later execution of T3, if the original node is accessed through some other pointers, the access checks can easily translate those pointers to point to 0xB by looking at the heap prefix and the x-th mapping entry. Similarly, if the node starting at 0xB is about to be accessed by a pointer, the access check code will confirm the access to be valid by simply looking at the prefix of this node. After committing T3's result, the local copy of the node will be deallocated and T3's status byte and meta byte in the prefix of the original node will be reinitialized to zero (shown on the right).

In summary, there are two main advantages of using heap-prefix to implement access checks. First, the status byte can tell the access checks whether or not a node has been copied. Second, the meta-data bytes allow the speculative thread to find the mapping entry in O(1) time, which speeds up the process of address translation for copy-in operations.

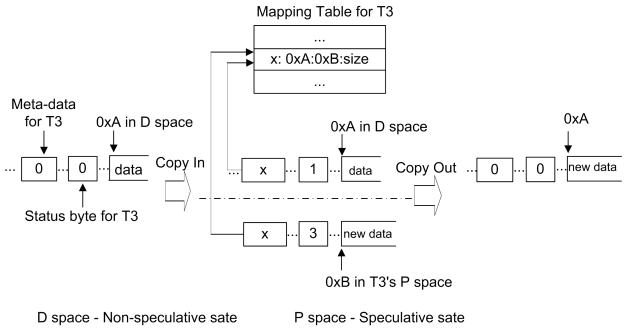

Figure 6. An Example Of Heap Status Transition.

2.4.2 Implementing Misspeculation Checks

To determine if speculation is successful, misspeculation checks have to be performed. The main thread maintains a version number for each variable in a version table. When a speculative thread uses a variable, it makes note of the variable's current version number. When the results of a speculative thread are to be committed to non-speculative state, misspeculation check is performed by the main thread. The main thread ensures that the current version number of a variable is the same as it was when the variable was first used by the speculative thread. If no mismatch is found, the speculation is considered as successful. Otherwise, misspeculation occurs and the result will be discarded. This is because speculative thread must have prematurely read the variable. This method worked effectively for array variables and scalar variables [35, 36].

In a program using dynamic data structures, however, the number of nodes in the structure can be very large, and hence the version table can become very large. Consequently, searching a node in the version table can impose large runtime overhead. Now that we have the heap prefix that can tell how the node is being used by other threads at any time, we can exploit this information to perform the misspeculation check for the dynamic data structure without using a version table.

The key idea of our approach is that when the main thread checks the result of thread i, it also checks if any other thread is using a node modified by thread i. If so, that thread's execution will fail as it is working on an incorrect speculatively read value. This method works because the main thread commits results of speculative threads' in a sequential order.

```
1:  if (spec[i] == FAIL)
2:     return FAIL;
3:  for each node mapping entry e in mapping[i] {
4:     for each thread j's status on e.addr_non-spec s[j] {
5:        if (s[j] == ALREADY_COPIED)
6:           spec[j] = FAIL;
7:        if (s[j] == ALREADY_READ
             and meta-data[j] == taskID[j])
8:           spec[j] = FAIL;
9:     }
10: }
11: return SUCCESS;
```

Figure 7. Misspeculation Checks For Heap Objects.

Fig. 7 shows our algorithm. For each node mapping entry e in the mapping table of thread i, the main thread examines other thread's status byte of the node starting at *e.addr_non-spec*. If another thread's status byte is ALREADY_COPIED, then speculation of that thread fails (line 5-6). Note that status ALREADY_COPIED means the node has been modified and hence has an entry in the mapping table. When the node is copied back, the status byte and meta-data byte for thread i is reset to zero.

If the main thread finds that another thread j's status byte is ALREADY_READ, then situation may be more complex because the status ALREADY_READ can be set during the current work assigned to thread j or during a previously assigned work to thread j. The latter case happens if in an earlier iteration, thread j only read this node. Therefore, there was no entry in the mapping table and hence the status byte and meta-data byte were not cleared. However, these two cases can be distinguished using the *task ID* stored in the meta-data byte j. The main thread only needs to check if the meta-data byte j's value is equal to thread j's current task ID. If they are the same, then thread j's execution is marked as failed.

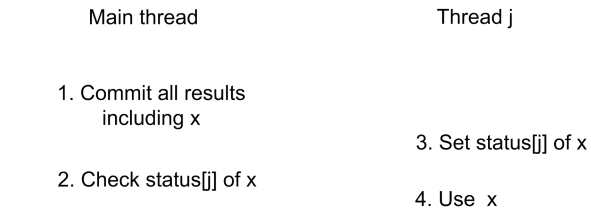

Figure 8. A Possible Data Race During Misspeculation Check.

Note that there exists a data race between checking thread j's status byte and setting the byte. However, this data race is harmless as it does not affect correctness. This is because we require the main thread to commit the result before checking other threads' status bytes and the speculative thread to update the status byte before accessing the node. Fig. 8 shows an example where step 2 and 3 are clearly racing against each other. If step 2 reads the value after step 3, then thread j's execution will be marked as failed which is correct, because thread j may read the old value of x (step 4 happens before step 1). If step 2 reads the value before step 3, thread j's execution will not be marked as failed. This is also correct because thread j is using the latest value of x.

In summary, the advantage of using heap-prefix in implementing misspeculation checks is that status bytes is used to identify any two threads that are accessing the same node. This eliminates the requirement of maintaining a version number of each node.

2.4.3 Discussion On Meta-data Bytes

For each thread, we choose to use one byte to store the meta-data, i.e., the index of the mapping entry or the *task ID*. Since one byte can at most hold 256 numbers, using one byte may impose some limitations in certain situations and hence needs to be discussed.

First, if the meta-data byte of a thread is used to store mapping entry indexes, the mapping table size must have less than 256 entries to avoid overflow. This means for each task, the number of modified node should be less than 256. In some cases, this assumption may not be true. If a mapping table overflow occurs, the corresponding task should be considered as failed to ensure the correctness. However, having too many overflow events means that using one byte is not enough and performance loss results. To solve this problem, we can profile the program to find out how many nodes are modified in each task on average and choose multiple bytes to store the indexes for each thread if necessary.

Second, if the *task ID* is stored in one byte, the number may also wrap around and cause a problem in a very extreme case. Specifically, when thread i writes a node at iteration a and thread j reads the same node at iteration b where $b \equiv a \pmod{256}$ and it never uses the node after that, we may incorrectly mark thread j as failed. However, even if this extreme case arises, the correctness is not affected at all – a false misspeculation is reported and the computation is unnecessarily repeated.

2.5 Double Pointers

As described earlier, the need for address translation can be reduced by using copy-on-write scheme. The address translation is needed only for a copied node. It can be done by using the status byte and meta byte during the copy-in operation. However, we need to efficiently perform the address translation for a pointer during a copy-out operation. This is because nodes being copied-out may have many pointer fields.

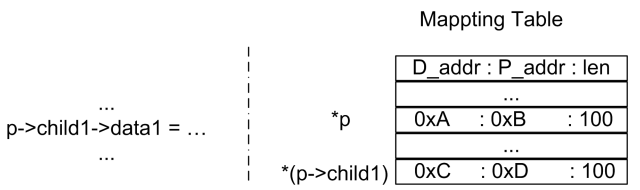

Figure 9. Internal Pointer.

Fig. 9 shows a simple example involving two nodes pointed to by pointers *p* and *p→child1*. The node **p* has already been copied in from non-speculative address 0xA to speculative address 0xB as shown in the mapping table. Now when the assignment is about to execute, the node **(p→child1)* will be copied from 0xC into 0xD. The value of *(p→child1)* will also change to 0xD so that the assignment takes effect on the local copy starting at 0xD. At the time of committing results to non-speculative state, the main thread must scan the mapping table to copy these two nodes back to the non-speculative state. However, the value in *(p→child1)* is still 0xD and of course it needs to be translated to 0xC. To do this, one way would be to locate the field by adding it to the mapping table, and then comparing the value in this pointer with all P_addr in the mapping table. This process entails significant overhead in programs using dynamic data structures as all nodes are linked into the structure through pointers.

To tackle the above problem, we present an augmented pointer representation – *double pointers*. For each pointer variable *p*, the compiler will allocate 8 bytes. 4 bytes for the non-speculative state address (denoted by *p.D_addr*) and 4 bytes for speculative state address (denoted by *p.P_addr*). When a node is allocated by the main thread and pointed to by *p*, its starting address is stored in *p.D_addr*. When a node is created by a speculative thread and pointed by a pointer *p'*, the starting address of this node is stored in *p'.P_addr*. If the node is created as a part of the copy-in operation, we set *p'.D_addr* to be *p.D_addr* (assuming *p'* is the local copy of *p*). Otherwise, we set *p'.D_addr* to be *p'.P_addr*. For any reference of a pointer *p*, if it is in the main thread, then *p.D_addr* will be used; otherwise *p.P_addr* will be used. For a pointer assignment *p = q*, all 8 bytes will be copied.

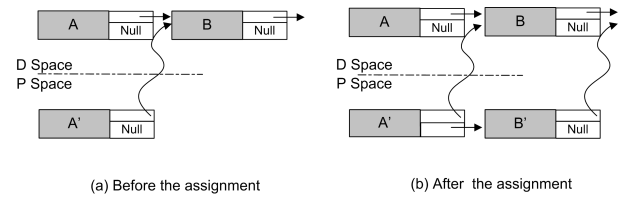

Figure 10. Double Pointer.

With this scheme, we can easily resolve the problem shown in Fig. 9. Consider now the illustration in Fig. 10 where A denotes **p* and B denotes **(p→child1)*. As we can see in Fig. 9(a), before the assignment, A's local copy keeps B's non-speculative address in the *D_addr* field. Its *P_addr* field has been set to NULL. After the assignment, a local copy of B has been created and pointed by the P_addr field of pointer *p→child1*. When we copy these two nodes back, we do not have to go to the mapping table for address translation. Instead, we can directly copy them back, as A still holds B's address in the non-speculative state.

The double pointers scheme also ensures the correctness when the shape of a dynamic data structure changes due to the update of some pointer fields in the computation. Let us consider the example in Fig. 3 again where three possible pointer related changes are encountered.

(Changing the Shape) If the branch at line 2 is taken, the node pointed to by *tmp* will be moved into the front of the buffer and pointed by *head*. Fig. 11 shows the process of this shape transformation under our scheme. As shown in Fig. 11(a) Before executing line 4, the parallel thread has two local pointers *prev'* and *tmp'*, which are the copies of pointers *prev* and *tmp* respectively. Since the statement at line 4 updates the node pointed to by *prev'* (node B), a local copy of node B is created through the access check, and the *P_addr* field of *prev'* points to this copy. After line 4, the *next* pointer field of node *B'* contains the address of node *D*.

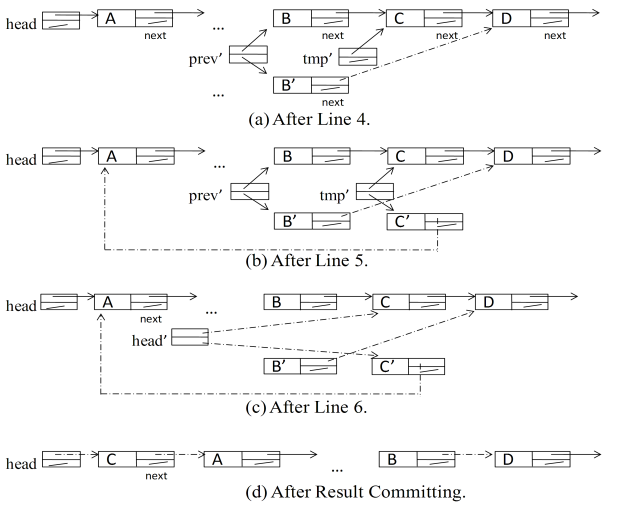

Figure 11. Changing The Shape Of Dynamic Data Structure.

When executing line 5 as shown in Fig. 11(b), a local copy of node *C* is created and pointed by the *P_addr* field of pointer *tmp'*. After this statement, all 8 bytes of the original pointer *head* are copied into the *next* pointer field in node *C'*. Thus, the *D_addr* field in *next* contains the address of node *A*. The statement at line 6 creates a local copy of pointer *head*. As shown in Fig. 11(c), after copying the contents of pointer *tmp'*, its *D_addr* field has the address of node *C* and *P_addr* has the address of node *C'*. Finally, copy-out operations in the result-committing stage will change the content of pointer *head* and the *next* pointer field of node B and C. The updated pointer is represented by the dash line in Fig. 11(d), which reflects the update to the LRU buffer.

(Adding A New Node) If the branch at line 9 is taken, a new node is allocated and the least recent used node is deallocated. Fig. 12 shows the process of adding a new node. In Fig. 12(a), a new node N is created by a parallel thread and pointed by a local pointer *n'*. After line 14, the *next* field of this new node is the same as the *head* pointer whose *D_addr* bytes are storing the address of node A. After line 15, the parallel thread changes the content of *head* pointer by creating *head'* and making its both *D_addr* and *P_addr* fields store the address of node N (see Fig. 12(b)). When copy-out operation, the main thread can recognize the new node by checking if the two fields of *head'* are the same. After the copying operation, pointer *head* is pointing the new node N which is now in the front of the LRU buffer (see Fig. 12(c)).

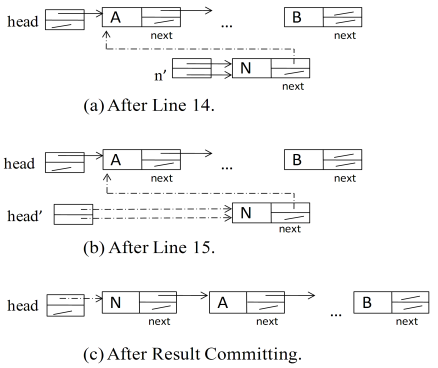

Figure 12. Adding A New Node Into A Dynamic Data Structure.

(Deleting A Node) Fig. 13 shows the process of executing lines 16-18, which deallocates the last node in the LRU buffer. As shown in Fig. 13 (a), after executing line 16, a local pointer *m'* is created by a parallel thread and pointing to the second last node B. When *free* is called on node C at line 17, the parallel thread simply marks the node as deallocated in the mapping table instead of actually call the function. This is because node C is in the non-speculative state and it cannot be deallocated until the speculative computation performed by this parallel thread is decided to be correct. After line 18, a local copy of node B is created because its pointer fields is speculatively set to *NULL*. In the result-committing stage, the main thread will actually deallocate the node C based on the mark in the mapping table and the *next* pointer field (8 bytes) of node *B* is also set to *NULL* due to the copy-out operation.

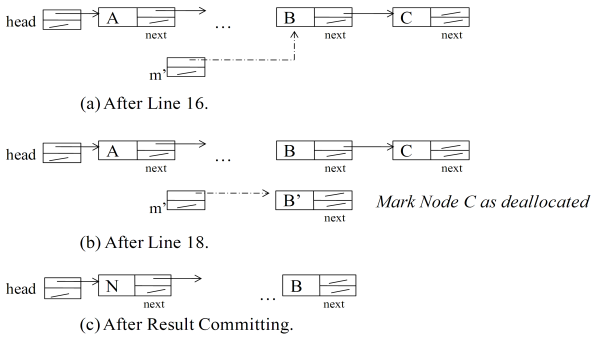

Figure 13. Deleting A Node From A Dynamic Data Structure.

2.6 Techniques And Their Benefits

Challenges	Copy-on-write Scheme	Heap Prefix	Double Pointers
Copying Operation Overhead	Reduced	-	-
Mapping Table Access Overhead	Reduced	Reduced	-
Misspeculation Check Overhead	-	Reduced	-
Address Translation Overhead (Copy-in)	Reduced	Reduced	-
Address Translation Overhead (Copy-out)	-	-	Reduced

Table 1. Techniques And Their Benefits.

In this section, we introduced three techniques to address the overhead and address translation problems when speculatively parallelizing a program using dynamic data structures. The three techniques were: copy-on-write, heap prefix, and double pointers. Table 1 summarizes the benefits of these techniques.

3. Other Optimizations

3.1 Eliminating Unnecessary Checks

An access check precedes each write access that is performed to the heap in speculative state. Although implementing access checks via heap prefix can greatly reduce their overhead, the overhead of access checks can be still significant due to the frequency with which they are performed. Therefore, in this section, we develop additional compile-time optimizations for eliminating access checks.

3.1.1 Locally-created Heap Objects

When a node is created by a speculative thread, it will have a valid speculative state address. Therefore, accesses performed to a locally created node do not require access checks. The algorithm is shown in Fig. 14 provides simple compile-time analysis to identify accesses that are guaranteed to always access locally created nodes. For each basic block, we first identify pointers that hold an address returned from a *malloc* function call. Then we track propagation of these pointers to other pointer variables and thus identify additional accesses that do not require an access check. In the analysis, each pointer assigned by *malloc* is placed into the *GEN* set. If a pointer is assigned with a pointer that is not holding a local heap address, it is placed in the *KILL* set. Then we compute the *IN* set for every basic block in a control flow graph based on the equations shown in this figure. Given the *IN* set of a basic block, it is easy to determine whether or not to introduce an access check before a pointer dereference.

Initialize IN (B_0) = $\emptyset$;

OUT(B) ={IN(B) - KILL(B)} $\cup$ GEN(B);
IN(B) = $\bigcap_{P \in pred(B)}$ OUT(P);
where
GEN(B) = $\{p : \exists\, p = malloc(...)$ in $B\}$
$\cup \{p : p = q$ where $q \in$IN(B) or GEN(B) $\}$;
KILL(B) = $\{p : p = r$ where $r \notin$
IN(B) and GEN(B) $\}$;

Figure 14. Locally Created Heap Objects.

3.1.2 Already-copied Heap Objects

Given a reference to a node, if we are certain that there is an earlier write which caused the node to be copied, then we do not need an access check for the reference. The analysis required for this optimization is quite similar to the analysis described in the preceding optimization. The difference is that the *GEN* set contains pointers through which a write is performed to a node instead of pointers assigned by *malloc*.

3.1.3 Read-Only Heap Access

If, following initialization, a node is only read throughout the execution, then it is impossible for such a node to cause a misspeculation. Therefore, access checks are not required for such nodes at all. However, it is challenging to identify such nodes at compile time due to pointer aliasing. Specifically, the same memory address may be pointed to by two or more pointers at runtime. During compile time, even if we identify that the access to a node through one pointer is always a read, the node may still be modified through another pointer at runtime.

Fortunately, there has been much research work on alias analysis. For any two pointers, the alias analysis responds with three possible answers, "yes", "maybe" and "no", indicating they do or maybe or do not point to the same location. We can take advantage of such analysis to conservatively identify read-only nodes. In particular, any two pointers with answer "yes" or "maybe" from alias

analysis, are considered as aliases. Next, we can perform the analysis shown in Fig, 15. For any access to a node through a pointer in $ReadOnlySet$, we do not insert any access checks, because these accesses must involve read-only nodes.

```
Initialize ReadOnlySet(S) = {all pointer variables};

for each pointer p {
  if there is a write access to the address held in p {
    ReadOnlySet(S) = ReadOnlySet(S) - {p};
  }
}
```

Figure 15. Finding Read-Only Heap Object.

3.2 Optimizing Communication

In our original CorD model [35, 36], the communication between the main thread and speculative threads was implemented through expensive system calls such as read and write to pipes. Use of pipes strictly prevents speculative threads from accessing the non-speculative state memory, as values required by parallel threads are passed through pipes. This mechanism works fine in [35] because few values need to be communicated at runtime. However, the same is not true in this work – many more values are communicated to speculative threads even when copy-on-write is used. Thus, overhead of using pipes to pass values is high.

In this paper, we relaxed the restriction of the state separation by allowing a speculative thread to directly read the non-speculative state memory. This results in highly efficient communication. We also use busy-waiting algorithms to synchronize threads because they achieve low wake-up latency and hence yield good performance on a shared memory machine [23].

4. Experiments

4.1 Experimental Setup

To show the effectiveness of our techniques, we use 7 benchmarks from the LLVM test suite and SPEC2000 that make intensive use of heap based dynamic data structures. In Table 2 the first two columns give the name and the description of each program. The next column shows the type of dynamic data structure used and the last column shows the original source of the program. The rest of the programs in these benchmark suites were *not used* due to one or more of the following reasons:

- they do not contain parallelizable loops;
- even though they contain parallelizable loops, no speculation is required for parallelization; or
- they contain parallelizable loops but seldomly use dynamic data structures, and thus can be efficiently parallelized using other techniques [5, 34–36].

Similar to the previous works on speculative parallelization, we parallelize loops for which the number of loop-carried dependencies is below a threshold number.

Name	Description	Dynamic Data Structure	Original Source
BH	Barns-Hut Alg.	Tree	Olden
MST	Mininum Spanning Tree	Tree, hash	Olden
Power	Power pricing	Graph, hash	Olden
Patricia	Patricia trie	Tree, hash	Mibench
Treesort	Tree sorting	Tree	Stanford
Hash	Hash table	List, hash	Shootout
Mcf	Vehicle scheduling	List, graph	Spec2000

Table 2. Dynamic Data Structures Benchmarks.

In our experiments, we first use Pin [15] instrumentation framework to profile loops in these programs with a smaller input. Then the runtime dependences are analyzed and regions that are good candidates for speculative parallelization are identified. Next the LLVM [20] compiler infrastructure is used to compile these programs together with the analysis result and our parallelization template, so that the sequential version can be transformed into the parallel version. The parallelization template contains the implementation of our runtime system including thread creation, interaction, mapping table, misspeculation check etc. During the transformation of the code, access checks are inserted preceding each heap access. The profiling is performed for a small input and the experimental data is collected by executing parallelized programs on a large input. All our experiments were conducted under Fedora 4 OS running on a *dual quad-core* (i.e., 8 cores) Xeon machine with 16 GB memory. Each core runs at 3.0 GHz.

4.2 Performance

We first compare the execution time of a program between its sequential version and parallel version. In this experiment, all the techniques and optimizations are used. We observed that running these programs under the original CorD model [35] led to at least 2x slowdown of parallel versions over sequential versions regardless of the number of speculative threads. However, with our proposed techniques, significant speedup is obtained. Fig. 16 shows the execution speedup of these programs for varying number of speculative threads created by the main thread.

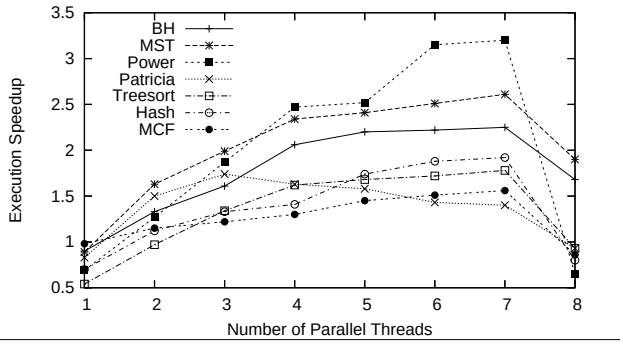

Figure 16. Performance On An 8-core Machine.

Our results show that for all programs except *Patricia*, the speedup continues to increase as the number of speculative threads is increased from 1 to 7. In particular, the highest speedup of *Power* is 3.2, and that of other programs is between 1.56 and 2.5 when 7 speculative threads are used. In the case of *Patricia*, we found that significant amount work has to be done sequentially by the main thread. Also, every speculative thread needs to copy up to 1 MB memory during execution. When more threads are used, more memory is copied. This can cause L2 cache pollution and hence dramatically affect the performance. For *Patricia* best speedup of 1.74 is achieved when we use three speculative threads.

Since we have a total of 8 cores, when 8 speculative threads are used in addition to the main thread, the speedup decreases. Except for *MST* and *BH*, all programs actually have a slowdown. The reason is due to the use of busy-wait synchronization. When 8 speculative threads and one main thread are run on an 8 core machine, context switch is required. However, using busy-wait constructs makes each thread to aggressively occupy a core. This leads to even worse performance. It is worth noting that using pipe will not have this problem as the main thread will be descheduled by OS. According to our experiments, however, the use of pipe causes the parallel execution to be much slower than the sequential one regardless of the number of parallel threads.

We also observe that using one speculative thread is slower than the sequential version. In the case of *Treesort*, even using two

threads cannot obtain any speedup. This behavior can be attributed to the overhead introduced by our model that more than nullifies the limited parallelism benefits.

We also measured the misspeculation rate of each execution. The highest rate we observed is 10.2% for *mcf* when using 7 parallel threads. This benchmark also has a large sequential portion. These two factors make the highest speedup of this program only 1.56. For other programs, the misspeculation rate is less than 1%. Thus, misspeculations have little impact on the performance.

4.3 Overhead Analysis

4.3.1 Time Overhead

We classify the execution time into 4 categories: *communication* (time spent on busy-waiting constructs), *access checks*, *misspeculation check followed by copy-out operations*, and *computation*. For speculative threads, we measured these times and averaged them across the threads. The experiment was conducted for 2, 4, and 7 threads. The results are shown in Fig. 17.

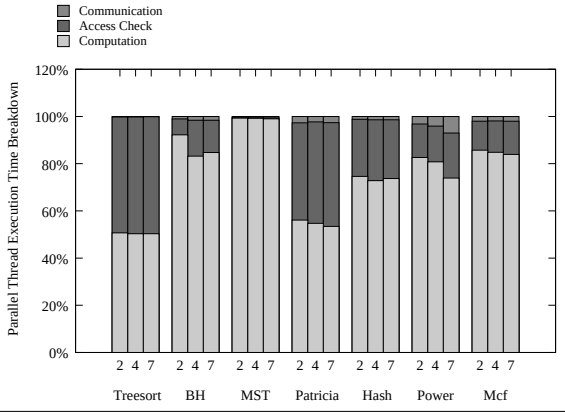

Figure 17. Time Breakdown: Spec. Threads.

From the figure, we can clearly see that regardless of the total number of speculative threads, each thread, on an average, spent from at least 50% (*Treesort*) to nearly 100% (*MST*) of the time on the computation. For some benchmarks like *Treesort* and *Patricia*, a significant amount of time is spent on access checks. The communication time is very low for all benchmarks, i.e. these threads do not spend much time on waiting for their work.

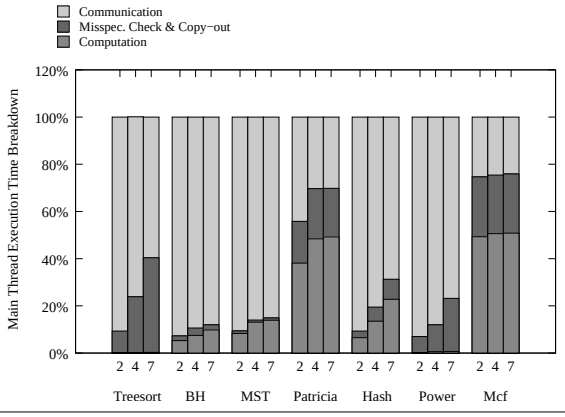

Figure 18. Time Breakdown: Main Thread.

Fig. 18 shows the execution time breakdown for the main thread, which is responsible for assigning work to speculative threads, performing misspeculation checks followed by copy-out operations, and executing the sequential part of the program. As we can see, for all programs except for *Patricia* and *Mcf*, the communication dominates the main thread's execution, which means the main thread is waiting for the results of speculative threads most of the time. During the rest of the time, the main thread does more work on misspeculation checks and copy-out operations for *Treesort* and *Power*, and more work on sequential computation for *Patricia*, *Mcf*, *BH*, *MST* and *Hash*. For the last 3 programs, the sequential computation portion becomes larger as the number of parallel threads increases from 2 to 7. This is because the total execution time is reduced due to greater parallelism and hence the sequential part becomes a greater fraction of the total execution time.

4.3.2 Space overhead

While the parallel execution is faster, it requires more memory space as each node has extra bytes for heap prefix and double pointers and each thread needs its own space. Therefore, we conducted an experiment to measure the peak value of memory consumption of the parallelized program for varying number of threads. Fig. 19 shows the results.

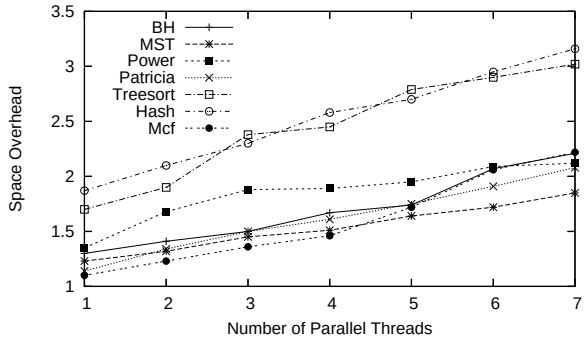

Figure 19. Space Overhead.

As we can see, the memory consumed by the parallel version of all programs is between 1.1x and 3.2x compared to the sequential version. Note that for most benchmarks except for *Treesort* and *Hash*, the space overhead caused by heap prefix and double pointers is at most 50% (when 7 threads are used) and often less than 20%. This is because each node in these programs takes over 60 bytes with about 2 to 6 pointers. Other space overhead mostly comes from the coping operations. Thanks to the copy-on-write scheme, only a small number of nodes need to be copied and hence the total overhead is not very large. For *Treesort* and *Hash*, however, the node size is only 12 bytes. Therefore, the double pointer scheme and heap prefix cause significant space overhead, especially when more threads are used as shown in the figure.

4.4 Effectiveness of Optimizations

As unnecessary access checks can be eliminated by the proposed analysis, we compared the number of static and dynamic access checks with and without optimizations. Table 3 shows the number of eliminated checks and the total number of checks without any elimination. From this table, we first observe that a small number of static access checks lead to millions of dynamic checks. This is because access checks are inserted inside loops that have millions of iterations. We can also see that, on an average, our optimization eliminates 69.5% of static access checks which correspond to 71.5% of dynamic access checks. So without the optimization, each thread may waste significant number instructions at runtime on performing unnecessary checks.

4.5 Comparison with Transactional Memory

Transactional memory (TM) [1, 3, 11, 12, 24, 27, 29] has been an active area of research. It is designed to enforce the atomicity of shared memory accesses in parallel programs and cannot be

Program Name	Checks Eliminated	
	Static	Dynamic (Million)
BH	5/7 (71.4%)	0.55/0.67 (82.1%)
MST	7/11 (63.6%)	8.9/13.5 (65.9%)
Power	55/60 (91.6%)	9.5/10.8 (88.0%)
Patricia	53/66 (80.3%)	62.4/79.7 (78.3%)
Treesort	3/6 (50%)	72.7/143.2 (50.7%)
Hash	7/12 (58.3%)	312.8/463.3 (67.5%)
Mcf	20/28 (71.4%)	968.2/1418.9(68.2%)
Average	69.5%	71.5%

Table 3. Effectiveness Of Eliminating Access Checks.

directly used to parallelize sequential programs [22]. However, since it has the capability of tracking dependences and detecting dependence violations between two transactions, we conducted an experiment to see the performance of using software based TM (STM) in the speculative parallelzation work.

In this experiment, we manually transform the program such that each task is put into a transaction and every access to the potentially shared memory in a task is monitored by the TM system. Similar to [22], we also add the explicit synchronizations into transaction functions to enforce the in-order commit, This is important for maintaining the sequential program semantics. The TM implementation we used is based on a state-of-art algorithm - Sun's Trasactional Locking 2 (TL2) [4]. Table 4 shows the speedup comparisons between our approach and STM-based solution when 2, 4 and 7 parallel threads are used respectively.

Programs	2 threads		4 threads		7 threads	
	Ours	STM	Ours	STM	Ours	STM
BH	1.33	0.59	2.06	0.68	2.25	0.73
MST	1.63	0.84	2.34	0.94	2.61	1.02
Power	1.27	0.93	2.47	0.97	3.20	1.08
Patricia	1.50	0.43	1.63	0.52	1.40	0.47
Treesort	0.97	0.34	1.62	0.41	1.78	0.40
Hash	1.12	0.64	1.41	0.73	1.92	0.87
Mcf	1.15	0.54	1.30	0.58	1.56	0.61

Table 4. Speedup Comparisons.

From the table, we can see that using STM in speculative execution has slowdowns in most cases. Only for *Power* and *MST*, a slight speedup can be achieved when 7 parallel threads are used. Our results are consistent with [22] which also shows that STM typically nullifies the performance gains in compiler parallelized sequential applications. There are several reasons for the performance loss. First, STM needs special mechanisms to avoid or resolve deadlock and live-lock situations. Second, STM aims to achieve good throughput and fairness. This requires STM to consider the priorities of transactions [32]. Besides, STM internally uses locks to prevent data races [4] and barriers to ensure strong atomicity [28, 30] and in-order commit [22]. These special considerations are not necessary for speculative parallelization. Instead, they result in high runtime overhead for STM while providing a convenience for programmers writing parallel applications.

Note that Mehrara et al. [22] propose customized STM for speculative parallelization. Their work assumes dependent variables can be identified at compile time and thus they use a set of special registers to track such variables. However, for the programs using dynamic data structures, cross-iteration dependences cannot be recognized statically. Therefore, their work is not applicable for the class of programs we consider.

5. Related work

The TLS technique is useful in improving the performance of sequential code. While there has been much research on TLS, it is hardware based requiring redesigned architectures [7, 21, 31, 37] or non-trivial modifications to the existing architectures such as special buffers [10, 25, 26], versioning cache [9], or versioning memory [8]. However, the hardware features are not present in any commercial multicores and this makes software speculation attractive as it can be used on existing machines.

Recently, software based TLS techniques have been proposed [5, 14, 17–19, 35]. Although all of them can be implemented purely in software, they use different schemes to handle speculative execution. The work proposed in [5, 14, 34–36] are all based upon the realization of state separation. In particular, the result of speculative computations are stored in a separate space instead of the non-speculative state. If misspeculation occurs, the result is simply discarded. Otherwise, the result is merged into the non-speculative state. State separation in [5, 14] is achieved by using different address spaces (process-based) and hence entails larger copying overhead compared to the scheme using the same address space (thread-based) [35]. Although OS-assisted techniques using paging hardware [2, 5, 14] can be used to implement copy-on-write at page level, the false-sharing problem leads to excessive misspeculations. To tackle this problem, the prior work [5] proposed to allocate each potentially shared variable on a separate page. However, this approach is impractical for heap based dynamic data structures. The reason is that such data structures may contain millions of nodes and hence, it is impossible to allocate one memory page for every node. Our other work on CorD [34–36] was also not practical for programs using dynamic data structures. In this paper we developed techniques that enable CorD to support dynamic data structures.

Kulkarni et al. proposed another speculative scheme [17–19] that allows the non-speculative state to be overwritten by speculative results. Once a misspeculation occurs, the original state can be recovered by performing the reverse computation of the speculative one. However, this scheme is only applicable for the programs using work lists. Besides, users need to use special constructors, mark all *commute* functions, which can be executed in any order, and define the reverse computation of each *commute* function in their programs. This places much burden on the users. In contrast, our method is profiling based and the transformation is performed by the compiler automatically.

Recently, Johnson et al. [13] and Du. et al. [6] presented different compiler algorithms on decomposing a sequential instruction stream into multiple speculative threads. In other words, they focus on how to identify and construct parallel threads from a sequential program, and hence their work is complimentary to our work.

6. Conclusion

For programs using heap based dynamic data structures, speculative parallelization is challenging. This is because the size of the dynamic data structure can be very large, moving heap data between non-speculative state and speculative state can be expensive, and address translation of accesses to data structure fields is needed. We proposed techniques and optimizations that effectively address these challenges. Our experiments show maximum speedups from 1.56 to 3.2 on **a real machine** for a set of programs that make extensive use of heap based dynamic data structures.

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] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, 2006.

[2] D. Bovet and M. Cesati. Understanding the linux kernel. 2005.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, 2006.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th Intl. Symp. on Distributed Computing*.

[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] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI '04: Proceedings of the 2004 ACM SIGPLAN conference on Programming language design and implementation*, pages 71–81, 2004.

[7] 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.

[8] 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.

[9] 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*, 1998.

[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, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

[12] 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.

[13] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of the 2004 ACM SIGPLAN conference on Programming language design and implementation*, pages 59–70, 2004.

[14] 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.

[15] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[16] 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.

[17] 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.

[18] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 233–243, 2008.

[19] 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.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04: Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

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

[22] 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.

[23] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[24] 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. *SIGOPS Oper. Syst. Rev.*, 40(5):359–370, 2006.

[25] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. *SIGARCH Comput. Archit. News*, 29(2):204–215, 2001.

[26] C. G. Quiñones, C. Madriles, F. J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, 2005.

[27] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: Proceedings of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[28] 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*, 2008.

[29] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[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: Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[34] 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.

[35] 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.

[36] 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.

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