Speculative Optimizations for Parallel Programs on Multicores

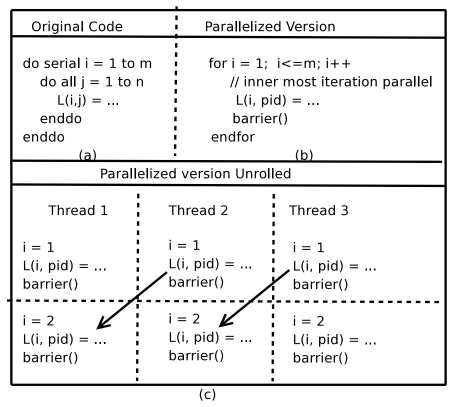
Vijay Nagarajan and Rajiv Gupta

University of California, Riverside, CSE Department, Riverside CA 92521
{vijay,gupta}@cs.ucr.edu

Abstract. The advent of multicores presents a promising opportunity for exploiting fine grained parallelism present in programs. Programs parallelized in the above fashion, typically involve threads that communicate via shared memory, and synchronize with each other frequently to ensure that shared memory dependences between different threads are correctly enforced. Such frequent synchronization operations, although required, can greatly affect program performance. In addition to forcing threads to wait for other threads and do no useful work, they also force the compiler to make conservative assumptions in generating code.
We analyzed a set of parallel programs with fine grained barrier synchronizations, and observed that the synchronizations used by these programs enforce interprocessor dependences which arise relatively infrequently. Motivated by this observation, our approach consists of creating two versions of the section of code between consecutive synchronization operations; one version is a highly optimized version created under the optimistic assumption that no interprocessor dependences that are enforced by the synchronization operation will actually arise. The other version is unoptimized code created under the pessimistic assumption that interprocessor dependences will arise. At runtime, we first speculatively execute the optimistic code and if misspeculation occurs, the results of this version will be discarded and the non-speculative version will be executed. Since interprocessor dependences arise infrequently, misspeculation rate remains low. To detect misspeculation efficiently, we modify existing architectural support for data speculation and adapt it for multicores. We utilize this scheme to perform two speculative optimizations to improve parallel program performance. First, by speculatively executing past barrier synchronizations, we reduce time spent idling on barriers, translating into a 12% increase in performance. Second, by promoting shared variables to registers in the presence of synchronization, we reduce a significant amount of redundant loads, translating into an additional performance increase of 2.5%.

1 Introduction

The advent of multicores presents a promising opportunity for exploiting fine grained parallelism present in programs. Low latency and higher bandwidth for inter-core communication afforded by multicores allows for parallelization of codes that were previously hard to parallelize. Programs parallelized in the above fashion typically involve threads that communicate via shared memory, and synchronize with each other *frequently* to ensure that shared memory dependences between different threads are correctly enforced. In such programs threads often execute only hundreds of instructions independently before they have to synchronize with other threads running on other cores.



```
do serial i = 1 to m
   do all j = 1 to n
      L(i,j) = ...
   enddo
enddo
```

```
for i = 1;  i<=m; i++
     // inner most iteration parallel
      L(i, pid) = ...
     barrier()
endfor
```

```
Thread 1          Thread 2          Thread 3

i = 1             i = 1             i = 1
L(i, pid) = ...   L(i, pid) = ...   L(i, pid) = ...
barrier()         barrier()         barrier()

i = 2             i = 2             i = 2
L(i, pid) = ...   L(i, pid) = ...   L(i, pid) = ...
barrier()         barrier()         barrier()
```

Program.	Time at Barrier	Redundant Loads (%)	Enforced Dep. (%)
Jacobi	25	6.25	25
Cholesky	61	24.8	6
Recurrence	42	88.5	12.5
Equake	32	2.1	3.2
Swim	24	0	3.1
Bisort	18	3.2	9
MST	28	8.4	34

Fig. 1. Dependences enforced by synchronization and its characteristics

Fig. 1(a) shows a simple example of a sequential program with a doubly nested loop. While the iterations of the innermost loop are independent (*do all*) of each other, each iteration of the outermost loop (*do serial*) is dependent on values computed in the previous iteration. This naturally gives rise to a parallel implementation as shown in Fig. 1(b), where iterations of the innermost loop are distributed and computed by parallel threads, after which the threads synchronize at the barrier. Fig. 1(c) shows the loop-unrolled version of the parallel code, in which the arrows show the inter-thread dependences that are enforced by the barrier synchronization. As we can see from the above example, the purpose of adding the synchronization is to *enforce shared memory dependences across threads.*

(Effect of frequent synchronization) Such frequent synchronization operations, although required, can greatly affect program performance. In addition to causing the program execution to spend significant time at synchronization points waiting for other threads, synchronization operations also force the compiler to make conservative assumptions in generating code; for instance, variables cannot be allocated to registers across synchronization operations without precise analyses [16, 17] that guarantees that those variables that were allocated to registers cannot be modified by other threads. In other words, variables that are potentially *shared* cannot be allocated to registers across synchronization operations. To evaluate the importance of these factors, we analyzed the executions of a set of parallel programs that exploit fine grained parallelism as shown in the table in Fig. 1. As we can see from column 1, each program spends a significant portion of its execution time (as high as 61%) waiting at barrier synchronizations. Furthermore, immediately following synchronization, each processor executes significant number of redundant loads[1] (as shown in Fig. 1), owing to the fact that shared variables could not be allocated to registers because of synchronization.

(Infrequent Dependences) One interesting property which we observed in our study is that barrier synchronizations used by these programs enforce interprocessor dependences that arise relatively infrequently. For each load executed by a processor following barrier synchronization, we determined if the value that it read was generated by another processor prior to barrier synchronization. We found that only 6% to 35% of loads executed at each processor involved interprocessor dependences (column 3 in the table in Fig. 1). Motivated by this observation, our approach consists of creating two versions of the section of code between consecutive synchronization operations. One version is a highly optimized version created under the optimistic assumption that none of the interprocessor dependences that are enforced by the synchronization operation will actually arise. The other version is unoptimized code created under the pessimistic assumption that interprocessor dependences will arise. At runtime, we first speculatively execute the optimistic code and if misspeculation occurs, the results of this version will be discarded and the non-speculative version will be executed. Clearly, the efficacy of this approach hinges on the misspeculation rate. Since interprocessor dependences arise infrequently, as we saw in our study, misspeculation rate remains low. This allows us to execute the optimized code most of the times, leading to performance savings.

(Efficient misspeculation detection) Another important parameter that affects the performance is the efficiency with which misspeculation is detected. It is worth noting that there is a misspeculation when there is actually an interprocessor dependence. To detect misspeculation, we utilize the support for *data speculation* in the form of *Advanced Load Address Table* (ALAT) already present in itanium processors [1, 6] and adapt it for performing speculative optimizations for parallel programs. We expose this architectural support to the compiler via two new instructions: the speculative read `S.Rd` and the jump on misspeculation `jflag` instruction. The `S.Rd` instruction enables us to specify the range of memory addresses that are speculatively read, which are efficiently remembered in the ALAT. Once the speculative read `S.Rd` is executed, the hardware ensures that remote writes to any of the addresses, from other processors, invalidate the corresponding entry in the ALAT and set the misspeculation flag. The compiler is given the ability to read this flag via the `jflag` instruction and hence can react to misspeculation by jumping to the pessimistic non-speculative version.

We utilize this scheme to perform two speculative optimizations to improve parallel program performance. First, by speculatively executing past barrier synchronizations, we reduce time spent idling on barriers, translating into a 12% increase in performance. Second, by promoting shared variables to registers in the presence of synchronization operations, we show that we can reduce a significant amount of redundant loads, translating into an additional performance increase of 2.5%.

[1] For this study, redundant loads refers to the variables that had to be re-loaded after a barrier, expressed as a percentage of total number of loads in the program

2 Support for misspeculation detection

(ALAT for data speculation) The support for misspeculation detection is based on data speculation support already present in itanium processors [1, 6]. This hardware support is in the form of a hardware structure known as the *Advanced load address table* (ALAT) and the special instructions associated with it. When a data speculative load is issued with a special load instruction known as the *advanced load* `ld.a` instruction, an entry is allocated in the ALAT, storing the target register and the loaded address. Every store instruction then automatically compares its address against all entries of the ALAT. If there is a match, the corresponding entry in the ALAT is invalidated. Using the `chk.a` instruction, a check is performed before the speculatively loaded value is used; while a valid entry means that load speculation was successful, an invalid entry means that the data has to be reloaded. Our observation that the above HW support can be used for misspeculation detection in our scheme stems from the fact that entries in the ALAT are also invalidated by *remote writes to the same address from other processors* [1]. Thus, in our optimistic speculative version, if we load all values using advance load `ld.a` instructions, each load will create an entry in the ALAT. A subsequent remote write in another processor to any of the addresses loaded will precisely indicate an interprocessor dependency that we failed to enforce through our speculation. Such remote writes invalidate the ALAT entries and thus serve as a mechanism for misspeculation detection.

(Modified ALAT for misspeculation detection) However, there are some significant differences between data speculation, which is primarily meant for scalars in sequential code, and speculative optimization for parallel programs. First, speculatively loading all values in the optimized optimistic version, via the `ld.a` instruction would very likely exhaust all possible entries in the ALAT. To deal with this size limitation, we propose the speculative read `S.Rd` instruction, through which the compiler can specify ranges of addresses that are to be read speculatively. In our design, each processor's ALAT can hold 4 such address ranges. We provide this capability so that vectors that are read speculatively can be specified succinctly by the compiler. If the compiler is unable to determine the range, it then can generate code with the `S.Rd` accompanying the loads in program without specifying the range. The hardware then takes the responsibility of inserting the address into any of the ranges maintained. The hardware does this conservatively – at any time the range of addresses maintained by the hardware is guaranteed to contain all the addresses read speculatively.

Another important semantic difference is that, while data speculation requires that local stores invalidate ALAT entries, speculation for parallel programs does not require this. Accordingly, the ALAT entries created by `S.Rd` instruction are not invalidated by local stores. It is worth noting that in the modified ALAT, there is still support for conventional data speculation. Thus local stores are made to invalidate ALAT entries for the advanced load instruction `ld.a` instruction, like before. Finally, we add a *flag* to the ALAT which is used to indicate misspeculation. This flag can be reset by the compiler using the `rflag` instruction. Whenever a remote write to one of the ALAT (ranges) is detected, the *flag* is set. The compiler can access this flag via the jump on misspeculation instruction, `jflag`, using which the compiler can jump to the pessimistic non-speculative version on detecting a misspeculation.

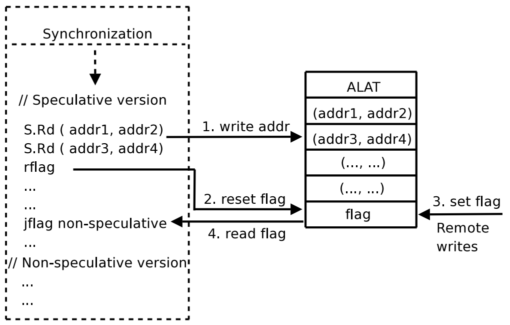
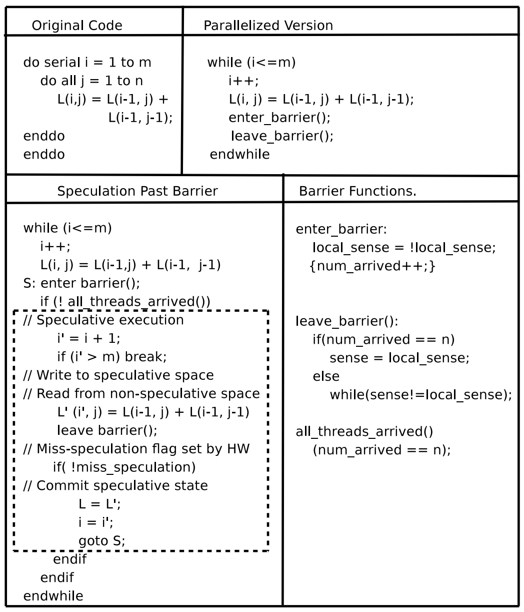


Fig. 2. Interaction of instructions on ALAT

Fig. 2 shows how the instructions interact with the ALAT. Following a synchronization operation, we first execute the optimistic version which assumes the absence of interprocessor dependences. The compiler specifies the range of addresses that the speculative code reads, via the `S.Rd` instruction. Accordingly, the ranges of addresses are remembered in the ALAT (step 1). The `rflag` instruction is then used to clear the misspeculation *flag* (step 2). While the optimistic version is executing, if there are any remote writes to any of the address ranges in the ALAT, the misspeculation flag is set (step 3). Finally at the end of speculative version, the compiler checks for misspeculation via the *jflag* instruction, and jumps to the non-speculative version, if there is a misspeculation.

3 Speculation Past Barriers



Original Code

```
do serial i = 1 to m
   do all j = 1 to n
      L(i,j) = L(i-1, j) +
               L(i-1, j-1);
enddo
enddo
```

Parallelized Version

```
while (i<=m)
    i++;
    L(i, j) = L(i-1, j) + L(i-1, j-1);
    enter_barrier();
    leave_barrier();
endwhile
```

Speculation Past Barrier

```
while (i<=m)
   i++;
   L(i, j) = L(i-1,j) + L(i-1,  j-1)
S: enter barrier();
   if (! all_threads_arrived())
// Speculative execution
      i' = i + 1;
      if (i' > m) break;
// Write to speculative space
// Read from non-speculative space
      L' (i', j) = L(i-1, j) + L(i-1, j-1)
      leave barrier();
// Miss-speculation flag set by HW
      if( !miss_speculation)
// Commit speculative state
         L = L';
         i = i';
         goto S;
      endif
   endif
endwhile
```

Barrier Functions.

```
enter_barrier:
   local_sense = !local_sense;
   {num_arrived++;}


leave_barrier():
   if(num_arrived == n)
      sense = local_sense;
   else
      while(sense!=local_sense);

all_threads_arrived()
   (num_arrived == n);
```

Fig. 3. Speculative execution past barrier

Barrier synchronization is commonly used in programs that exploit fine grained parallelism – threads often execute only hundreds of instructions before they have to wait for other threads to arrive at the barrier. A thread that arrives at a barrier first, does no useful work until other threads arrive at the barrier and this amounts to the time lost due to the barrier synchronization. In order to reduce the time lost due to the barrier synchronization, compilers typically try to distribute equal amounts of work to the different threads. However, threads often do not execute the same code and this in turn causes a variation in the arrival times. Moreover, even if each thread executes the same code, they can each take different paths leading to a variation in number of instructions executed. From our experiments, we found that the time spent on barrier synchronization can be as high as 61% of the total execution time for the set of programs considered. To reduce the time spent idling on synchronization, we propose a compiler-assisted technique for speculating past barrier synchronizations. Our technique is based on the observation that inter-thread shared memory dependences that the barriers strive to enforce can be infrequent. By speculatively executing past a barrier, we in turn are speculating that the inter-thread dependences do not manifest during speculation. When the inter-thread dependences are infrequent, more often than not our speculation succeeds and we are able to achieve significant performance improvements. We illustrate our approach using

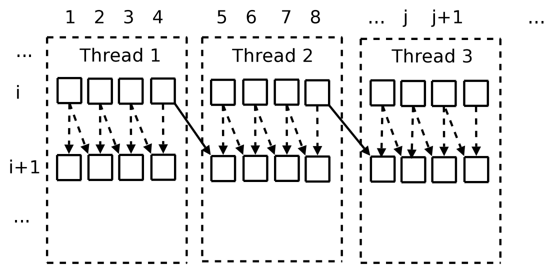
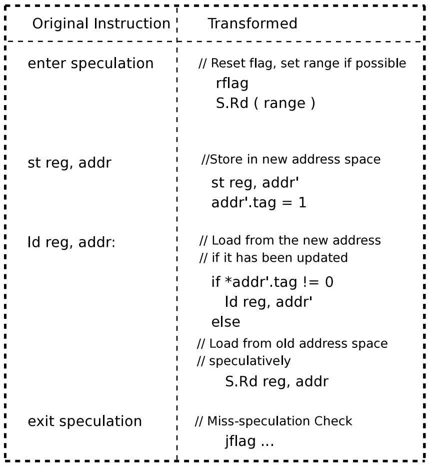


Fig. 4. Dependences exercised

an example (Fig. 3) which shows the original sequential code, the unoptimized parallelized version and our optimized version which shows the compiler transformation for speculatively executing past barriers. The sequential code shows a doubly nested loop: each iteration of the inner loop can be done in parallel (*do all*), while the outer loop has to be performed sequentially since there is a loop-carried dependence. While each iteration of the inner loop could be given to different thread, this is not done typically [19]. To increase the computation-to-communication ratio and to preserve locality, each thread is given a part of the vector as illustrated in Fig. 4, where each thread is given a chunk consisting of four elements. Consequently in every iteration, each thread computes the values in its chunk, reading values computed from the previous iteration, after which it synchronizes with other threads by entering the barrier. We enable speculation past the barrier by generating code as illustrated in Fig. 3. Once a thread arrives at a barrier, we announce that the current thread has arrived at the barrier as shown in the function *enter-barrier*. Then we check if all threads have reached the barrier; if so, then there is no need to speculate and we move on to the next iteration. However, if not all threads have reached the barrier, we proceed to execute speculatively past the barrier.

(Thread Isolation) We create a safe address partition for the speculative thread to work on. The primary benefit of this isolation is that *name* dependences that manifest between the speculative and the non-speculative threads can be safely ignored, and do not cause a misspeculation. Moreover, we do not require heavy-weight rollback in case of a misspeculation; we merely discard the newly created address space as in our prior work [21]. If the speculation is successful, then the speculative state is merged with the non-speculative state. It is important to note that the above tasks viz. thread isolation, recovery from misspeculation and committing the results of a successful speculation are performed in software by the compiler. The compiler ensures thread isolation by writing to a separate address space during speculation. In other words, stores are transformed to store into the separate speculative address space. However, this creates a potential problem for reads; reads need to be able to read from original or new address space, depending on whether the read address has already been written into. To deal with this, each word of the new address space is associated with meta data which is initialized to 0. Whenever there is a store to a word, the meta-data for the corresponding word is set to 1 as shown in Fig. 5. Depending on the value of the meta-data, loads then read from the speculative (new) or non-speculative address space. However, for the programs considered, which essentially deals with loops working on vectors, the compiler is able to statically determine whether the reads have to read from the original or new address space, and this obviates the need for maintaining meta data for most loads and stores.

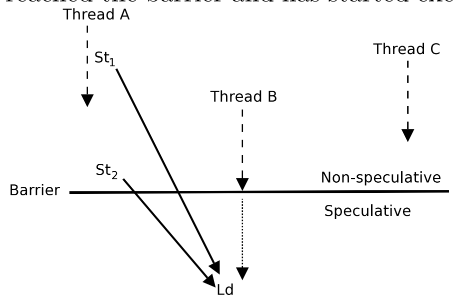


```
Original Instruction   Transformed

enter speculation      // Reset flag, set range if possible
                          rflag
                          S.Rd ( range )

st reg, addr           //Store in new address space
                         st reg, addr'
                         addr'.tag = 1

ld reg, addr:          // Load from the new address
                       // if it has been updated
                         if *addr'.tag != 0
                           ld reg, addr'
                         else
                       // Load from old address space
                       // speculatively
                           S.Rd reg, addr

exit speculation       // Miss-speculation Check
                           jflag ...
```

Fig. 5. Code transformation

(Misspeculation Detection) We use the modified ALAT to detect misspeculation. Upon entering speculative execution, we use the `rflag` instruction to reset the misspeculation flag. Then we set the range of addresses that are read using the `S.Rd` instruction. If the compiler is not able to statically determine the range of addresses read, then `S.Rd` instructions are made to accompany the loads as shown in Fig. 5 – the hardware will ensure that all the addresses that are read from, are remembered in the ALAT. Whenever there is a remote write to any of the addresses remembered in the ALAT, it would then invalidate the ALAT entry and set the misspeculation flag. Recall that a remote write to any of the addresses read speculatively, signals an interprocessor dependency which the barrier was attempting to enforce. However, it is also the dependency that was not enforced due to the speculation and hence such a dependency flags a misspeculation. In Fig. 4, the interprocessor dependences are indicated with solid arrows. Consequently, at the end of the speculation, we check for misspeculation flag and if it is not set, we commit the speculative state. Committing the state involves copying the contents of the newly allocated space into the original non-speculative address space.

(Reducing Misspeculation rate) Although the dependences enforced by the barriers are infrequent, they can still cause misspeculation if they manifest after the speculative code starts executing. As we can see from Fig. 6, thread B has reached the barrier and has started executing past it in speculative mode. When



```
ORIGINAL CODE:

do all j = (start) to (start + size)
   L(i,j) = ...
enddo
```

```
PROFILING:

do j = (start) to (start + size)
   if L(i,j) is shared
      SHARED = SHARED U {j}
enddo
```

```
TRANSFORMED:

do j = (start) to (start+size)
   if in SHARED
      L(i,j) =...
enddo

do j = (start) to (start+size)
   if j not in SHARED
      Li,j) =...
enddo
```

Fig. 6. Reducing Misspeculation rate.

it encounters the load instruction, St_2 (thread A) has not yet been executed. In other words, the dependence between the St_2 and Ld has not yet been enforced. Thus, when St_2 eventually executes in thread A, a misspeculation will be flagged. On the contrary, let us assume St_2 does not exist (or writes a different address) and so the only interprocessor dependence is between St_1 and Ld. In this case, note that by the time Ld instruction is executed in the (speculative) thread B, St_1 from thread A has already executed. In other words, the dependence between St_1 and Ld has already been enforced, and so this interprocessor dependency will not cause a misspeculation. Thus, to reduce the chance of misspeculation, it would be beneficial to advance writes to shared data (like St_1 and St_2), which is the focus of this optimization. To perform this optimization, we first identify the iterations that write to shared data as shown in the profiling step of Fig. 6. We then perform those iterations earlier than others. It is worth noting that we can perform this reordering only if the iterations in the inner loop can be performed in any order (do all).

4 Speculative Register Promotion

Recall that the purpose of synchronization operations are to enforce shared memory dependences across threads. However, lack of precise information about the

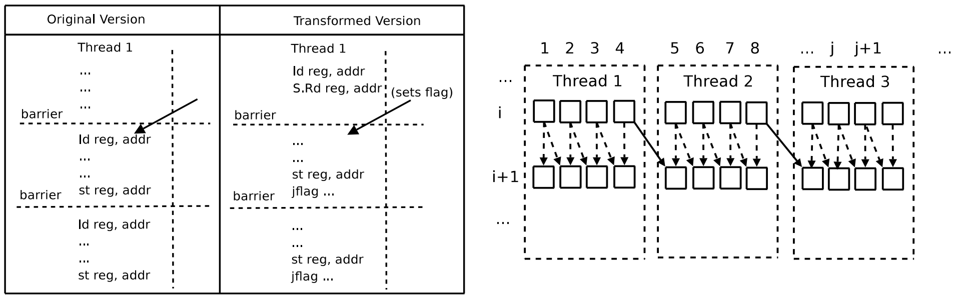


Fig. 7. (a) Redundant loads due to barriers (b) Data partitioning

dependences can lead to the execution of a significant number of redundant loads. In our study we found that a significant number of loads executed around synchronization operations were redundant loads. Fig. 7(a) illustrates the reason for these redundant loads. When a barrier is reached, there is a need to dump all the variables that have been allocated registers to memory. Likewise, when a thread leaves a barrier, all the dumped variables have to be reloaded into registers. This is because, without information that guarantees that a variable is local to the thread, the compiler can not allocate the variable to a register across synchronization operations. Let us consider the same example of the doubly nested loop, whose inner loop can be parallelized. Recall that each thread is given a part of the vector to work on, to increase locality. Since each thread accesses and updates parts of the vector, the vector as a whole is *shared*. As shown in Fig. 7(b), thread 2, for example, reads in $L[4]$ through $L[8]$ and writes $L[5]$ through $L[8]$, every iteration. Since the vector is shared, the compiler cannot allocate individual elements across synchronizations. Thus, it cannot allocate $L[4]$ to a register in thread 2, because each iteration it is written by thread1. However, note that elements $L[5]$ through $L[7]$ are actually exclusive to thread 2 and could be allocated to registers across synchronizations. Without this fine grained partitioning information, it is hard for the compiler [16, 17] to figure out which of the variables are local to threads. On the other hand, it is relatively easier to estimate which of the variables are shared or local, by using profiling. Thus, during profiling run, for each variable accessed by a thread, we determine if that variable is indeed modified non-locally by a different thread. If not, such variables are identified to be variables suitable for register promotion. We can then speculatively promote such (probably) local variables to registers, provided we can detect the case when our speculation is wrong. In this optimization, we use the `S.Rd` instruction to speculatively allocate the variable to a register, at the same time, remembering the address in the ALAT. Whenever there is a remote write to the same address, hardware sets the misspeculation flag, which helps us detect the situation when our speculation is wrong. As we can see in the transformed version in Fig. 7, by promoting the variable to the register, we are able to remove the redundant loads every iteration. It is worth noting that while the redundant load can be removed, we still have to store the value to the memory every iteration before the barrier. This would serve as a means to detect misspeculation in case the same variable had been promoted to a register in some other thread.

However, speculatively promoting registers is not as simple as removing the loads during speculation (past the barrier) and remembering the addresses in the ALAT. To see why, let us consider the Fig. 8.

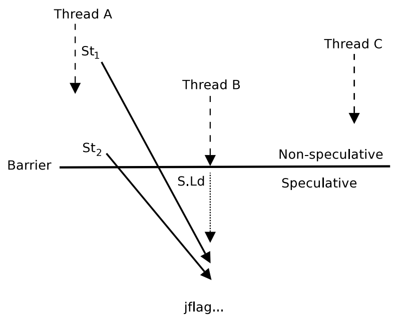
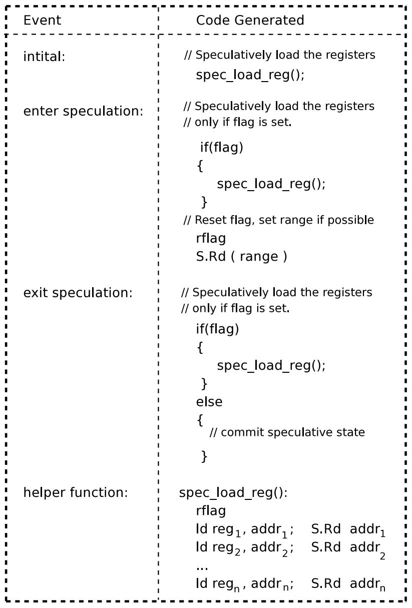


Fig. 8. Promoting registers during speculation

As we can see by executing $S.Rd$ we are remembering the addresses that have been speculatively promoted to registers. This will mean that, if there is a store in thread A (St_2), this will invalidate the ALAT entry in thread B and notify us of our misspeculation. However, let us consider the case of St_1, from thread A, which we assume to write to the same address. Since St_1 has already been executed in thread A, before the thread B has arrived at the barrier, we have no way of detecting this dependency; in other words, we cannot detect the misspeculation in this scenario.



```
Event                 Code Generated

intital:              // Speculatively load the registers
                        spec_load_reg();

enter speculation:    // Speculatively load the registers
                      // only if flag is set.
                        if(flag)
                        {
                            spec_load_reg();
                        }
                      // Reset flag, set range if possible
                        rflag
                        S.Rd ( range )

exit speculation:     // Speculatively load the registers
                      // only if flag is set.
                        if(flag)
                        {
                            spec_load_reg();
                        }
                        else
                        {
                          // commit speculative state
                        }

helper function:      spec_load_reg():
                        rflag
                        ld reg_1, addr_1;    S.Rd  addr_1
                        ld reg_2, addr_2;    S.Rd  addr_2
                        ...
                        ld reg_n, addr_n;    S.Rd  addr_n
```

Fig. 9. Code transformation

To handle this situation, we speculatively load the variables into registers once, initially (outside the loop) as shown in Fig. 9. This enables us to remember the addresses of the loaded variables in the ALAT. We also reset the misspeculation flag via the `rflag` instruction. Whenever a thread reaches a barrier, before speculatively executing past it, we check if the flag is set. If no flag is set, this means that there has been no stores to the speculatively promoted addresses. This in turn means that we can safely continue to use the promoted registers without loading. However, flag that has been set at this point means that there has been a store to one of the ALAT entries. This, in turn, means that we have to reload the variables into registers and this is precisely what we do. While reloading the variables to registers, we again use corresponding `S.Rd` to remember the loaded values in the ALAT. Having taken care of registers, we now proceed with the speculative execution. Likewise, before we exit the speculation, we again check if the misspeculation flag has been set. It is important to note that this could mean one of two things: either the values that have been speculatively loaded have been written into, or the values that have been speculatively promoted have been written into. To take care of the latter, we again load the variables into registers, at the same time, remembering the loaded addresses in the ALAT. If the misspeculation flag has not been set at this point, we commit the speculative state as usual.

5 Experimental Evaluation

Processor	8 processor, inorder
L1 Cache	32 KB 4 way
L1 hit latency	1 cycle
L2 Cache	512 KB 8 way
L2 hit latency	9 cycle
Memory latency	200 cycle
Coherence	MOSI bus based

Program.	Source
Jacobi	Iterative solver
Cholesky	Cholesky gradient
Recurrence	Linear recurrence
Equake	Earthquake simulation
Swim	Weather prediction
Bisort	Bitonic sort
MST	Minimum spanning tree

Fig. 10. (a) Architectural parameters used for simulation (b) Programs used

In this section, we present the results of an experimental evaluation of our scheme for speculation past barriers and speculative register promotion. First and foremost, we wanted to measure the performance increase we obtained by speculatively executing past barrier synchronizations. Since key to a good performance is low misspeculation rate, we also wanted to study the misspeculation rate and see how it is affected by our compiler transformation for reordering the loops. We also wanted to measure the additional performance increase we obtained by speculatively promoting variables into registers in the presence of synchronizations. But before we present our experimental results we briefly describe our implementation and the benchmarks used. We implemented the architectural support in the SESC [15] simulator, which is a cycle accurate CMP simulator targeting the MIPS architecture. This is because the proposed architectural support with the modified ALAT is not available in current processors and hence had to be simulated. For our simulation, we used the architectural parameters listed in table in Fig. 10. The benchmarks we used are a set of seven parallel programs listed in Fig. 10. Each of the programs frequently synchronize using barrier synchronizations, which makes them interesting subject programs for our study. *Cholesky* (kernel 2) and *Recurrence* (kernel 6) are parallelized versions of Livermore loops, whose implementations are described in [18]. *Equake* and *Swim* are from the SPEC *Openmp* benchmark suite, while *Bitonic sort* and *MST* are from the *Olden* benchmarks suite. Finally, we also used the parallelized version of the *Jacobi* iteration. We rewrote each program to make use of synchronization constructs associated with the simulator and compiled each program to run on the simulator using the simulator's cross compiler. It is important to note that since the above programs synchronize frequently using barriers, they are interesting subject programs for the evaluation of our technique.

(Execution Time Reduction) First, we measured the execution time reduction using the architectural and compiler support described. For this experiment, we used an ALAT of size 4 and allowed the hardware support to automatically maintain the ranges within the 4 entries. We generated code to remember the addresses in the ALAT (by using the S.Rd instruction) for the heap and global data, since stack addresses are local to the thread. We then used profiling to identify the part of shared data that is local to each thread and promoted such variables to registers. To keep misspeculation at a minimum we use the compiler technique described and reorder the iterations of loop, so that shared data is updated as early as possible. As we can see from Fig. 11(a), we are able to

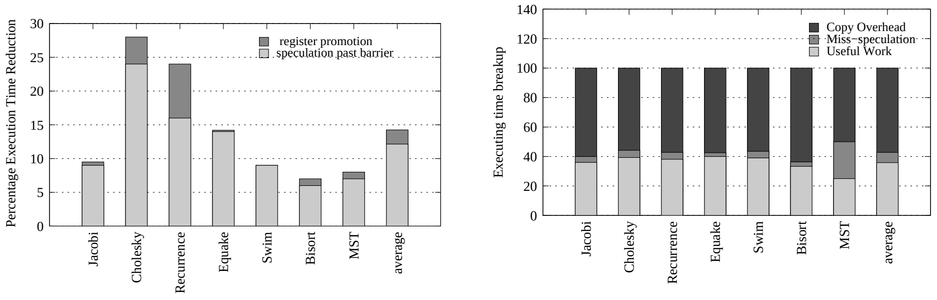


Fig. 11. Execution time reduction and break up

reduce the execution time significantly. The percentage reduction in execution times ranges between 6% (Bitonic sort) and 24% (Livermore loop 2). On an average we could achieve a 12% reduction in execution time by speculatively executing past barriers. By performing speculative promotion of variables into registers, we could achieve further reduction in execution times. *Recurrence* and *Cholesky*, have a significant number of redundant loads around synchronizations and for these programs we could reduce the execution times significantly – 8% and 4% respectively. On an average, execution time was reduced by a further 2.5% across all benchmarks, due to speculative register promotion.

To gain further insight as to why we were getting the speedup, we measured how the original time spent in synchronization (without speculation) was now being spent with speculation. As we can see from Fig. 11(b), about 37% of the original time spent in barrier is now channeled into performing useful work. We can also see that the time spent inside the handler recovering from misspeculation is relatively low (about 5%), owing to small number of misspeculations. However, significant time (about 58%) was spent performing copies for maintaining and committing speculative state.

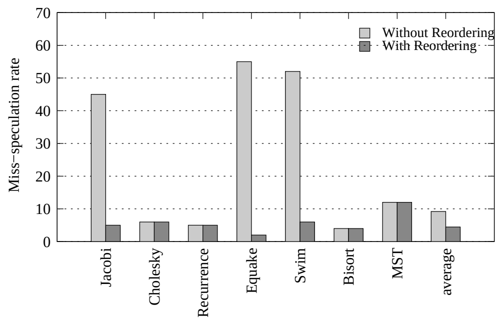


Fig. 12. Efficacy of reordering

(Efficacy of loop reordering) Recall that to reduce misspeculation we reordered the loops so that updates to shared data take place earlier. To determine the efficacy of this optimization we measured the misspeculation rates before and after application of this transformation. As we can see from Fig. 12, this optimization significantly reduces the misspeculation rates for Jacobi (44% to 5%), Equake(54% to 2.3%) and Swim (51% to 6%) programs. In the above three programs, there was a shared update in the end of each thread's execution which was causing misspeculation. Once we moved this earlier, the misspeculation rate significantly dropped.

6 Related Work

(Speculation past barriers) The *Fuzzy Barrier* [3] is a compile time approach to reduce time spent idling on barriers, by specifying a range of instructions over which the synchronization can take place instead of a specific point where the threads must synchronize. However, this approach relies on the compiler to find instructions that can be safely executed while a thread is waiting for others to reach the barrier. *Speculative lock Elision* [14] and *Speculative Synchronization*[9] are hardware techniques to speculatively execute threads past synchronization operations. While the former dynamically converts lock-based codes into lock-free codes, the latter also applies to flag synchronizations and barriers. The thread that has reached the barrier, speculatively executes past the barrier. Hardware support (addition of per block tags to the cache, modifications to the cache coherence, support for register checkpointing) is used to monitor dependence violations between the speculate thread(s) that are executing past the barrier and other non-speculative threads that are yet to reach the barrier. If such a violation is detected, the speculative thread is rollbacked to the synchronization point. Our approach, on the contrary, augments the relatively lightweight HW support in the form of ALAT, that is already available in processors[1, 6], to perform misspeculation detection.

(Speculative Parallelization) There has been recent research on software techniques for speculatively parallelizing sequential codes [2, 21, 5]. However, while the above work focusses on exposing coarse-grained parallelism in sequential codes, the focus of this work is on exposing fine-grained parallelism present in parallel codes.

(Transactional Memory) The problem of detecting cross-thread dependence violations at run time is known as *conflict detection* under *Transactional memory* (TM) [4] parlance. STM systems [8] instruments loads and stores with *read/write barriers* to detect conflicts. On the contrary, HTM systems [4] rely on hardware support (modifications to caches/cache coherence) to detect conflicts. Hybrid TMs[10, 20] use hardware to perform the simple and common case and rely on software support to handle the uncommon case. Recent proposals on hybrid TM have proposed hardware support for conflict detection. While *SigTM*[10] uses hardware signatures for conflict detection, *RTM* proposed the *Alert-on-Update* [20] mechanism which triggers a software handler when specified lines are modified remotely. In the current work, we modify existing HW support in the form of ALAT to provide conflict detection. By storing the addresses in the ALAT as ranges, we take advantage of program properties to efficiently store the read set in the ALAT. Furthermore, we use the cross thread dependence detection capability to implement speculative optimizations for parallel programs.

(Data Speculation) There has been significant work on hardware structures to enable data speculative optimizations [13, 6, 7]. While [13] proposes the *store-load address table (SLAT)* to enable speculative register promotion, [6] and [7] use the ALAT available in Itanium processors [1] to enable sophisticated compiler optimizations such as speculative *partial redundancy elimination (PRE)* etc. In this work, we utilize a modified ALAT to perform data speculative optimizations in the context of parallel programs running on multicores.

(Other Related Work) Neelakantam et al. [12] use *hardware atomicity* to implement speculative compiler optimizations. However, while they primarily implement compiler optimizations such as partial inlining and unrolling for increasing single threaded performance, we are concerned with compiler optimizations for increasing parallel program performance on multicores. We used *exposed cache events* [11] to perform a variety of monitoring tasks, including detecting misspeculation when speculating past barriers. In our current work, in addition to using the ALAT for detecting misspeculation detection, we also propose compiler optimizations to reduce misspeculation and a technique for speculatively promoting registers in the presence of synchronization.

7 Conclusion

In this work, we adapt existing architectural support for data speculation in the form of ALAT, to make it applicable for parallel programs running on multicores. We utilize this scheme to perform two speculative optimizations to improve parallel program performance. First, by speculatively executing past barrier synchronizations, we reduce time spent idling on barriers, translating into a 12% increase in performance. Second, by promoting shared variables to registers in the presence of synchronization, we reduce a significant amount of redundant loads, translating into an additional performance increase of 2.5%.

References

1. Itanium software developers manual. In: http://www.intel.com/design/itanium /manuals/iiasdmanual.htm
2. Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., Zhang, C.: Software behavior oriented parallelization. In: PLDI. pp. 223–234 (2007)
3. Gupta, R.: The fuzzy barrier: A mechanism for high speed synchronization of processors. In: ASPLOS. pp. 54–63 (1989)
4. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA (1993)
5. Kelsey, K., Bai, T., Ding, C., Zhang, C.: Fast track: A software system for speculative program optimization. In: CGO. pp. 157–168 (2009)
6. Lin, J., Chen, T., Hsu, W.C., Yew, P.C.: Speculative register promotion using advanced load address table (alat). In: CGO. pp. 125–134 (2003)
7. Lin, J., Chen, T., Hsu, W.C., Yew, P.C., Ju, R.D.C., Ngai, T.F., Chan, S.: A compiler framework for speculative analysis and optimizations. In: PLDI. pp. 289–299 (2003)
8. Marathe, V.J., III, W.N.S., Scott, M.L.: Adaptive software transactional memory. In: DISC. pp. 354–368 (2005)
9. Martínez, J.F., Torrellas, J.: Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In: ASPLOS. pp. 18–29 (2002)
10. Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: ISCA. pp. 69–80 (2007)
11. Nagarajan, V., Gupta, R.: Ecmon: Exposing cache events for monitoring. In: ISCA (2009)
12. Neelakantam, N., Rajwar, R., Srinivas, S., Srinivasan, U., Zilles, C.: Hardware atomicity for reliable software speculation. In: ISCA. pp. 174–185 (2007)
13. Postiff, M., Greene, D., Mudge, T.N.: The store-load address table and speculative register promotion. In: MICRO. pp. 235–244 (2000)
14. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO. pp. 294–305 (2001)
15. Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P.: SESC simulator (January 2005), http://sesc.sourceforge.net
16. Rinard, M.C.: Analysis of multithreaded programs. In: SAS. pp. 1–19 (2001)
17. Salcianu, A., Rinard, M.C.: Pointer and escape analysis for multithreaded programs. In: PPOPP. pp. 12–23 (2001)
18. Sampson, J., González, R., Collard, J.F., Jouppi, N.P., Schlansker, M., Calder, B.: Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In: MICRO. pp. 235–246 (2006)
19. Shirako, J., Zhao, J.M., Nandivada, V.K., Sarkar, V.: Chunking parallel loops in the presence of synchronization. In: ICS. pp. 181–192 (2009)
20. Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S., Scott, M.L.: An integrated hardware-software approach to flexible transactional memory. In: ISCA. pp. 104–115 (2007)
21. Tian, C., Feng, M., Nagarajan, V., Gupta, R.: Copy or discard execution model for speculative parallelization on multicores. In: MICRO. pp. 330–341 (2008)