Speculative Barriers with Transactional Memory

Manuel Pedrero, Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata

```
1: for (i ← 0; i < N; i ← i+1) do
2:     TX_START(tx)                          ▷ Not necessary when using check_spec
3:     for (j ← 0; j ≠ L*((tid+1)%2); j ← j+1) do
4:         dummy ← dummy+1
5:     end for
6:     TX_STOP(tx) / CHECK_SPEC(tx)
7:     SPEC_BARRIER(tx)
8: end for
9: LAST_BARRIER(tx)
```

Fig. 1. Barrier microbenchmark.

APPENDIX A
BENCHMARKS AND SPECULATIVE BARRIERS

The Barrier microbenchmark depicted in Fig. 1 represents a best-case scenario with no conflicts among barriers [1]. We used it to demonstrate that the proposed SB API does not introduce conflicts related to the implementation. The threads perform local computations in lines 3 to 5 and synchronize with a barrier afterwards, line 7. The condition in line 3 implies that only half of the threads actually work in each iteration of the outer loop, thus creating a load imbalance. In this scenario, the use of SPEC_BARRIER allows the idle threads to continue to the next iteration. As there is only a TX_STOP/CHECK_SPEC call between barriers, the speculation is limited to one barrier at a time.

Cholesky and Recurrence correspond with the second and sixth kernels of Livermore Loops (LFK) [2]. These kernels exploit fine-grain data parallelism with a high frequency of barriers [3]. The code for the LFK General Recurrence equation is shown in Fig. 2. We added a resizable inner chunk (lines 7 to 12) to support different transaction sizes. First, the kernel distributes the computation among threads in lines 1 to 3. The loop in lines 5 to 14 updates a single element in array w by accessing several elements of w and matrix b. This creates a recurrence due to read-after-write patterns in w, so a barrier in line 15 is mandatory at the end of each iteration of the outermost loop (lines 4 to 16). Nevertheless, dependencies in w are low, since only one w element is updated in each step, so it is a promising scenario to use SBs. The LAST_BARRIER in line 17 ensures that no deallocations are performed by speculative threads until the kernel computation finishes.

- *All authors are with the Department of Computer Architecture, University of Malaga, Spain, 29071.*
  E-mail: {mpedrero,quislant,eladio,zapata,oplata}@uma.es

Manuscript received November 22, 2019.

```
 1: ochunk ← N/#TH
 2: start ← tid*ochunk
 3: stop ← start + ochunk
 4: for (t ← 0; t ≤ N-2; t ← t+1) do
 5:     for (k ← start; k < stop; ) do
 6:         TX_START(tx)                     ▷ Not necessary when using check_spec
 7:         for (c ← 0; c < ichunk; c ← c+1) do
 8:             if (k < (N-t-1)) then
 9:                 w[t+k+1] ← w[t+k+1] + b[k][t+k+1]*w[t]
10:             end if
11:             k ← k + 1
12:         end for
13:         TX_STOP(tx) / CHECK_SPEC(tx)
14:     end for
15:     SPEC_BARRIER(tx)
16: end for
17: LAST_BARRIER(tx)
```

Fig. 2. Livermore loop 6 (Recurrence).

Fig. 3 shows an incomplete Cholesky factorization based on [3]. We added a resizable inner chunk (lines 15 to 19) to support different transaction sizes. The inner loop (lines 13 to 21) updates elements in the x array upper-half (line 17) by computing several elements from x and v arrays. A barrier in line 22 ensures that each iteration of the outermost loop (lines 3 to 23) accesses to updated elements of x. Even so, only a fraction of x is updated in each step, being a suitable case for SBs. SBs can be particularly useful, since the computation in the kernel decreases within each step of the outer loop, thus producing a load imbalance. The LAST_BARRIER in line 24 is used to protect subsequent deallocation of the arrays from being executed before the main loop has finished. CHECK_SPEC may be used in this kernel if we are certain that there are no dependencies in the inner loop, which is not trivial to analyze, but it is hinted in the original code found on Netlib [3].

DGCA [4], Fig 4, features a coloring graph algorithm which does not require complete information of the graph to compute the color of each node. Instead, a global array p holds a probability mass function of the color for each node and it is updated using the node's neighbors colors. A node may change its color according to p in each step. The barrier at line 25 ensures that all nodes access updated graph information in each step of the computation. Lines 4 to 26 comprise a single step, where each individual node checks for potential color collisions with its neighbors in lines 7 to 12. With this information, p is updated in lines 13 to 21, assigning maximum probability to the same color if no collisions are detected, or updating the color probability accordingly otherwise. Dependencies are generated in line

```
ii ← N
ipntp ← 0
repeat
    ipnt ← ipntp
    ipntp ← ipntp+ii
    ii ← ii/2
    i ← ipntp-1
    ochunk ← (ipntp-ipnt)/2 + (ipntp-ipnt)%2
    ochunk ← ochunk/#TH + ((ochunk%#TH)?1:0)
    i ← tid*ochunk
    end ← (ochunk*2*(tid+1)) + ipntp + 1
    start ← ipnt + 1 + (tid*2*ochunk)
    for (k←start; k < end; ) do
        TX_START(tx)                    ▷ Not necessary when using check_spec
        for (c ← 0; c < ichunk; c ← c+1) do
            i ← i+1
            x[i] ← x[k] - v[k]*x[k-1] - v[k+1]*x[k+1]
            k ← k + 2
        end for
        TX_STOP(tx) / CHECK_SPEC(tx)
    end for
    SPEC_BARRIER(tx)
until (ii > 1)
LAST_BARRIER(tx)
```

Fig. 3. Livermore loop 2 (Cholesky).

```
 1: p[N][#colors]                           ▷ Global color probability array
 2: start ← tid*N/#TH
 3: stop ← start + N/#TH
 4: for (t ← 0; t < T; t ← t+1) do
 5:     for (i ← start; i < stop; i ← i + 1) do
 6:         TX_START(tx)                    ▷ Not necessary when using check_spec
 7:         collision = 0
 8:         for (j ← 0; j < #adjacentNodes(i); j ← j+1) do
 9:             if (nodeColor(i) = nodeColor(j)) then
10:                 collision ← 1
11:             end if
12:         end for
13:         if collision then
14:             for (k ← 0; k < #colors; k ← k + 1) do
15:                 p[i] ← updateNodeColorProbCollision(i,k)
16:             end for
17:         else
18:             for (k ← 0; k < #colors; k ← k + 1) do
19:                 p[i] ← updateNodeColorProbNoCollision(i,k)
20:             end for
21:         end if
22:         updateNodeColor(i,p)            ▷ Potentially updates node color
23:         TX_STOP(tx) / CHECK_SPEC(tx)
24:     end for
25:     SPEC_BARRIER(tx)
26: end for
27: LAST_BARRIER(tx)
```

Fig. 4. Decentralised graph colouring algorithm (DGCA).

22, where the node color is either updated or not depending on *p*. Since only a decreasing fraction of the nodes will be changed in each step, it is a good scenario for SBs.

Stencil and Histogram are two benchmarks from the Parboil suite [5]. We took the OpenMP version of them and got a pthread alternative code for the kernels where the *omp parallel for* implicit barrier was replaced by an SB. Fig. 5 shows the Stencil kernel, an iterative Jacobi solver of the heat equation on a 3-D structured grid of dimensions *nx*,*ny* and *nz*. *A0* and *Anext* point to the global input array at first. Iterations are computed reading from *A0* and writing to *Anext*. After the barrier, each thread interchanges pointers to work with the updated data. Parameter T specifies the number of iterations and the number of barriers. With a small T, the barriers are few, and besides, the load is

```
function INDEX3D(nx, ny, i, j, k)
    return i + nx*(j + ny*k)
end function
chunk ← (nx - 2 + #TH - 1)/#TH;
start ← tid*chunk + 1;
stop ← start+chunk
for (t ← 0; t < T; t←t+1) do
    for (i ← start; i < stop; i←i+1) do
        for (j ← 1; j < ny - 1; j←j+1) do
            for (k ← 1; k < nz - 1; k←k+1) do
                TX_START(tx)
                Anext[INDEX3D(nx, ny, i, j, k)] ←
                    (A0[INDEX3D(nx, ny, i, j, k + 1)] +
                    A0[INDEX3D(nx, ny, i, j, k - 1)] +
                    A0[INDEX3D(nx, ny, i, j + 1, k)] +
                    A0[INDEX3D(nx, ny, i, j - 1, k)] +
                    A0[INDEX3D(nx, ny, i + 1, j, k)] +
                    A0[INDEX3D(nx, ny, i - 1, j, k)]) * c1 -
                    A0[INDEX3D(nx, ny, i, j, k)] * c0;
                TX_STOP(tx)
            end for
        end for
    end for
    SPEC_BARRIER(tx)
    temp ← A0;
    A0 ← Anext;
    Anext ← temp;
end for
LAST_BARRIER(tx)
```

Fig. 5. Stencil.

balanced, so the gains with SBs are little.

Fig. 6 shows the code for Histogram. It is a simple histogramming operation that accumulates the number of occurrences of each output value in the input data set. The output histogram (*histo*) is a two-dimensional matrix of char-type bins that saturate at 255. The Parboil input sets, from a particular application setting in silicon wafer verification, follow a roughly Gaussian distribution centered in the output histogram, thus causing a lot of conflicts. Parameter T defines the number of images processed. We added parameter C which defines an inner chunk for determine the size of the transaction. The kernel offers poor speculation opportunities as the histogram is reset at the beginning of the outer loop.

SSCA2 [6] is a graph theory application which consists of four kernels that require irregular access to a directed, weighted multi-graph. STAMP [7] includes a transactional parallel version of SSCA2 and focuses on kernel 1 due to its adequacy to TM. We use SBs in kernel 1 and 4 of SSCA2, which show a considerable amount of barriers. Kernel 1 constructs the graph from a tuple list by computing adjacency arrays for the vertices of a dense graph. Fig. 7 shows an excerpt of a thread computation from kernel 1. We can see barriers in lines 6 and 11 that should not be replaced with SB. The first barrier might be crossed speculatively, but is followed by dynamic memory allocation, which is not allowed inside transactions. The next barrier is not replaced because the code after the barrier uses the arrays allocated before the barrier, and a segmentation fault is risked. Consequently, both barriers are left as standard barriers, although we can use LAST_BARRIERs instead with the same result. We have used SBs in line 18 and within the *prefixSums()* function. This function has dynamic memory allocation and dependencies making it difficult to speedup the computation. This kernel offers few opportunities for SB

```
ochunk ← (imgHeight*imgWidth + #TH - 1)/#TH
start ← tid * ochunk
stop ← start + ochunk
chunkh ← (histoHeight*histoWidth + #TH - 1)/#TH;
starth ← tid*chunkh;
stoph ← starth + chunkh
for (t ← 0; t < T; t←t+1) do
  for (j ← starth; j < stoph; j←j+1) do
    histo[j] ← 0;
  end for
  SPEC_BARRIER(tx)
  for (j ← start; j < stop; ) do
    TX_START(tx)
    for (k ← 0; (k < C) ∧ (j < stop); k ← k + 1) do
      value ← img[j]
      if (histo[value] < 255) then
        histo[value] ← histo[value] + 1;
      end if
      j ← j + 1
    end for
    TX_STOP(tx)
  end for
  SPEC_BARRIER(tx)
end for
LAST_BARRIER(tx)
```

Fig. 6. Histogram.

```
[start,stop] ← createPartition(#edges, tid)
#vert ← #vertices(startVertex,start,stop)
xBegin()                                         ▷ Standard transaction
globMax#Vert ← max(globMax#Vert,#vert)
xCommit()
barrier()                                        ▷ Standard barrier
if (tid = 0) then
  outDegree ← dynamicAllocation(globMax#Vert)
  outVertexIndex ← dynamicAllocation(globMax#Vert)
end if
barrier()
[start,stop] ← createPartition(globMax#Vert,tid)
[outDegree,outVertexIndex] ← initialization(start,stop)
SPEC_BARRIER(tx)
outVertListSize ← fillArrays(outDegree,outVertexIndex)
LAST_BARRIER(tx)
prefixSums(outVertexIndex,outDegree,globMax#Vert)    ▷ Dynamic
  allocation and barriers inside
SPEC_BARRIER(tx)
TX_START(tx)
globOutVertListSize ← globOutVertListSize + outVertListSize
TX_STOP(tx)
LAST_BARRIER(tx)
...
```

Fig. 7. SSCA2 (Kernel 1). Compute Graph.

placing and it translates to performance.

SSCA2's kernel 4 divides the graph into highly interconnected clusters and minimizes the links among them. Fig. 8 sketches an excerpt of the kernel. The loop in line 4 determines the clusters in parallel. The barrier in line 8 might be a standard barrier as the code after it works with arrays filled in before, but we have tried an SB instead. The next barrier is suitable for SB, as the only dependency is with the previous *globCliqueSize* initialization by thread 0 in line 10. The following barrier should be a LAST_BARRIER since there are dependencies with *globCliqueSize* and *globIter*. After the loop, the partial cut lists are merged to proceed with the minimization of links among clusters. We left the barrier in line 30 untouched due to the dynamic memory allocation issue, whereas the barrier in line 33 can be a SB since only thread 0 has a dependency.

```
 1: ...
 2: vertVisited ← 0
 3: iter ← 0
 4: while (vertVisited < #vert ∨ iter < #vert/2) do
 5:   if (tid = 0) then
 6:     chooseVertToStart()
 7:   end if
 8:   SPEC_BARRIER(tx)
 9:   if (tid = 0) then
10:     globCliqueSize ← 0
11:   end if
12:   [cliqueSize,cutSetIndex] ← determineClusters(iter)
13:   SPEC_BARRIER(tx)
14:   if (tid = 0) then
15:     iter ← iter + 1
16:     globIter ← iter
17:   end if
18:   TX_START(tx)
19:   globCliqueSize ← globCliqueSize + cliqueSize
20:   TX_STOP(tx)
21:   LAST_BARRIER(tx)
22:   iter ← globalIter
23:   vertVisited ← vertVisited + globCliqueSize
24: end while
25: barrier()                               ▷ Merge partial cutset lists
26: if (tid = 0) then
27:   edgeStartCount ← dynamicAllocation(#THs)
28:   edgeEndCount ← dynamicAllocation(#THs)
29: end if
30: barrier()
31: edgeEndCount[tid] ← cutSetIndex
32: edgeStartCount[tid] ← 0
33: SPEC_BARRIER(tx)
34: if (tid = 0) then
35:   for (t ← 1; t < #TH: t ← t + 1) do
36:     edgeEndCount[t] ← edgeEndCount[t] + edgeEndCount[t-1]
37:     edgeStartCount[t] ← edgeEndCount[t-1]
38:   end for
39: end if
40: TX_START(tx)
41: globCutSetIndex ← globCutSetIndex + cutSetIndex
42: TX_STOP(tx)
43: LAST_BARRIER(tx)
44: ...
```

Fig. 8. SSCA2 (Kernel 4). Graph Clustering.

APPENDIX B
SUPER-LINEARITY IN RECURRENCE AND DGCA

Recurrence and DGCA show super-linear speedups for certain inputs. These codes exhibit poor cache reuse in the sequential execution due to irregular accesses to shared data. Such reuse is improved to a certain extent when adding hardware threads due to the partition of the global data structures among threads along with the increase in effective cache memory capacity, making this rare effect to appear. Fig. 9 shows the speedup and user cache misses collected from the simulator for these inputs. We have added a super-linearity metric which stands for:

$$Efficiency = \frac{Speedup}{Thread\,Count}$$

$$Super\text{-}linearity\,(\%) = 100(Efficiency - 1)$$

We show the super-linearity percentage on Recurrence and DGCA, where we can observe the abrupt drop in user misses as threads are added, highly correlated with the appearance of the super-linearity.

We can observe super-linear speedups for these benchmarks in the IBM Power8 architecture as well, but shifted

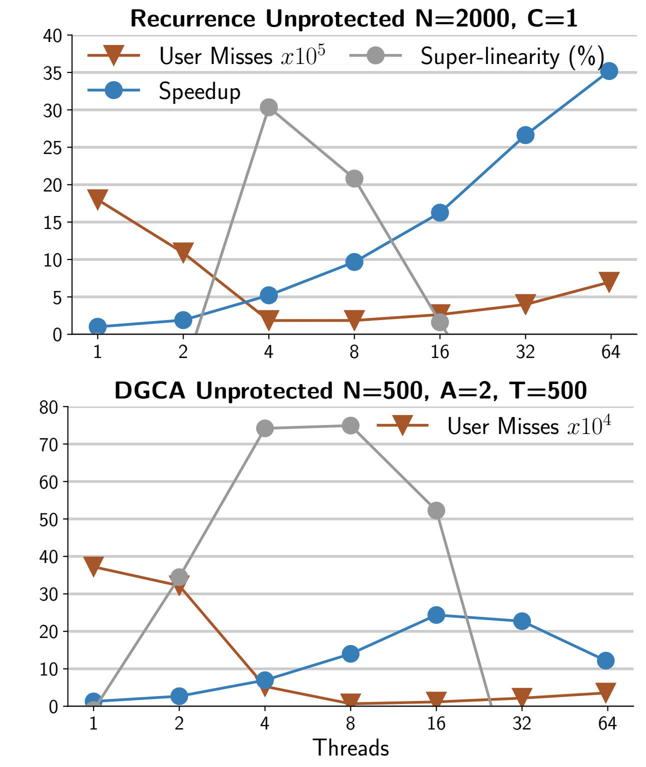

Fig. 9. RubyHTM user cache misses for Recurrence and DGCA Unprotected. A metric of super-linearity is shown

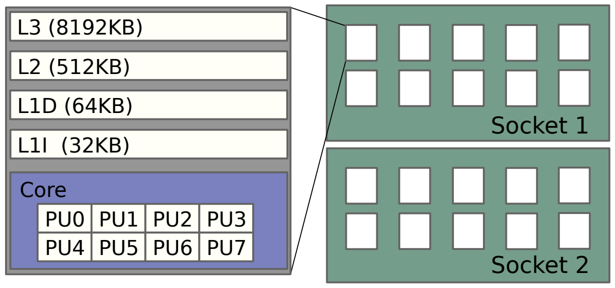

Fig. 10. IBM Power8 S822LC-8335 server configuration

because of the different configurations of inputs and architecture. It is worth to be noted the speedup jump when going from 8 to 16 threads that could be explained by the use of 1 and 2 sockets, respectively, where the latter doubles the shared L3 cache size (see the IBM Power8 server configuration in Fig. 10 and the Methodology for the thread affinity policy used in this paper). In the same way, the use of SMT (PU1-7) in experiments with 32-128 threads effectively reduces the amount of cache available for each thread, which also correlates with the performance achieved.

References

[1] L. Bonnichsen and A. Podobas, "Using Transactional Memory to Avoid Blocking in OpenMP Synchronization Directives," in *Int'l. Workshop on OpenMP*, 2015, pp. 149–161.

[2] J. T. Feo, "An analysis of the computational and parallel complexity of the Livermore Loops," *Parallel Computing*, vol. 7, no. 2, pp. 163–185, 1988.

[3] J. Sampson, R. Gonzalez, J. Collard *et al.*, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *Int'l. Symp. on Microarchitecture*, 2006, pp. 235–246.

[4] K. R. Duffy, N. O'Connell, and A. Sapozhnikov, "Complexity analysis of a decentralised graph colouring algorithm," *Information Processing Letters*, vol. 107, no. 2, pp. 60–63, 2008.

[5] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, 2012.

[6] D. A. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *Int'l. Conf. on High Performance Computing*, 2005, pp. 465–476.

[7] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Int'l. Symp. on Workload Characterization*, 2008, pp. 35–46.