MASTER THESIS

DUAL CONSISTENCY STORE BUFFER FOR OUT OF ORDER PROCESSORS

Student:
SINGH SAWAN
sawan.singh@studenti.polito.it

Supervisors:
ALBERTO ROS (UM)
aros@um.es
GUIDO MASERA (POLITO)
guido.masera@polito.it

CAPS
Computer Engineering
Department, UM Spain

VLSI Lab
Department of Electronics
and Telecommunications,
POLITO Italy

December 3, 2019

Acknowledgement

I would like to thank Prof. Alberto for allowing me to work with him. It was a pleasure to work with him. He provided a lot of motivation and guidance without which this work would be impossible to achieve. He was always there to help when I'm stuck with some problems. Apart from work he helped in every possible way to help me with my stay in Murcia.

I would also like to thank Prof. Masera at POLITO. He was very supportive when I approached him to be a supervisor from POLITO for my thesis. He is always so kind to me.

In the end, I would like to thank all the people in the lab I was working with Ashkan and Paco who made my time so comfortable.

SINGH SAWAN

Abstract

Store-buffer is an important part of a modern-day out of order processor. Store-buffer allows stores to retire in program order. The store is executed and moved to store queue from where they commit and then finally move to store-buffer, where they are written in the cache in program order. Once the store-buffer is full the store requests have to wait until a place is available in the store-buffer. This generates a stall thus not allowing other stores to enter the store-buffer. This gets worse if there is a miss at the head of the store buffer. Thus until the miss is resolved no operation in the store buffer is allowed to perform in the memory to maintain the program order. We address this problem in this work.

Our work leads to an approach where we can reorder the operations in the store buffer and perform them out of order without violating the program order. We took the help of compiler to help us identify the stores we can perform out of order and we designed a new store buffer with performs according to the information provided by the compiler. Other optimizations and configuration are also performed and checked to give a bigger picture.

Contents

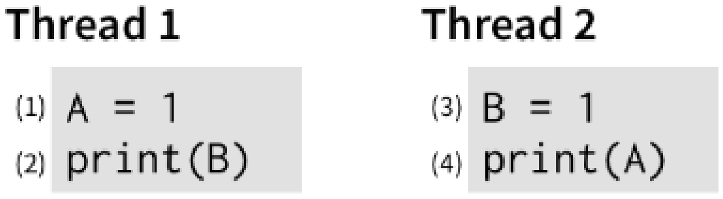
List of Figures

List of Tables

1 Background

1.1 Why this matters?

Memory consistency is a big problem that we are dealing with and still, we have not reached a stage where we can say we solved it. Memory consistency refers to a problem of how parallel threads use there shared space of memory with other threads. To explain it better let us take an example[1].



```
Thread 1              Thread 2

(1) A = 1             (3) B = 1
(2) print(B)          (4) print(A)
```

Figure 1: *Memory Consistency Example*

To understand what this program can output, we should think about the order in which its events can happen. Intuitively, there are two obvious orders in which this program could run:

(1)− > (2)− > (3)− > (4) : The first thread runs both its events before the second thread, and so the program prints 01.
(3)− > (4)− > (1)− > (2) : The second thread runs both its events before the first thread. The program still prints 01.
There are also some less obvious orders, where the instructions are interleaved with each other.

(1)− > (3)− > (2)− > (4) : The first instruction in each thread runs before the second instruction in either thread, printing 11.
(1)− > (3)− > (4)− > (2) : The first instruction from the first thread runs, then both instructions from the second thread, then the second instruction from the first thread. The program still prints 11. and a few others that have the same effect.

This is what we never want, we always want consistency behavior. This leads us to develop a memory model where the hardware promise programmer a particular execution order and the programmer promise to write programs keeping that order in mind.

At this stage, we are ready to discuss 2 main memory models. The sequential consistency and the Total Store Ordering (TSO)

1.1.1 Sequential Consistency

One nice way to think about sequential consistency is as a switch. At each time step, the switch selects a thread to run and runs its next event completely. This model preserves the rules of sequential consistency: events are accessing a single main memory, and so happen in order; and by always running the next event from a selected thread, each thread's events happen in program order.

The problem with this model is that it's terribly, disastrously slow. We can only run a single instruction at a time, so we've lost most of the benefit of having multiple threads run in parallel. Worse, we have to wait for each instruction to finish before we can start the next one—no more instructions can run until the current instruction's effects become visible to every other thread.

1.1.2 TSO

Modern processors contain a different layer of memories the closest to the core is the registers which are the fastest then we have an L1 Cache then L2 Cache and then L3 Cache. Generally, the memories close to the core are exclusive for each core and the memory far from the core is generally shared[2].

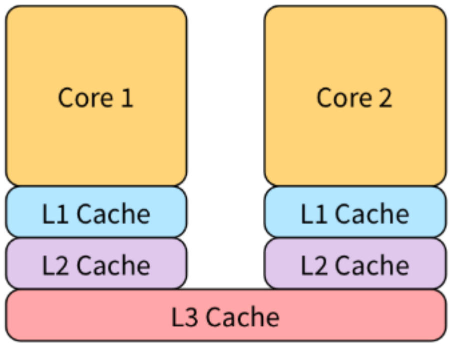
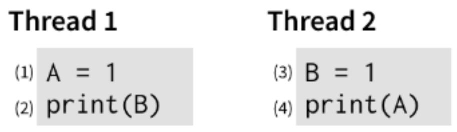


Figure 2: *Memory Hierarchy*

So for example, if we have the following operations to perform:-



Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

Figure 3: *Memory Hierarchy Example*

A = 1 is a long process as its a write and thus it has to go all the way till L3 which is a shared cache. So till then, the print(B) will have to wait. This is a huge drawback as a print(B) has nothing to do with the A = 1 and still, it has to wait.

One intuitive solution can be to store the result of A=1 in a local register and let the print(B) execute. This is the TSO memory model. The register is called the store buffer, All the work is done to develop this thesis is performed in the TSO memory model.

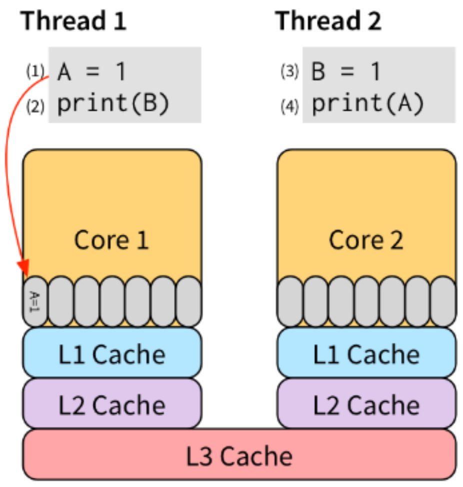


Figure 4: *TSO Memory Model*

So, the result is stored in a local register called store buffer which is a very fast operation thus the next operation which is print(B) does not have to wait much. Also if we have some other operation in place of print(B) which is dependent on A can look for the updated value of A in the store buffer. If the value is not found in the store buffer it goes to the lower memory hierarchy.

The problem with TSO is that the store buffers are exclusive for each core so if one other operation in some other thread for example in the figure 3 the print(A) will not be able to access the latest value of A as it is stored in the local store buffer which is not accessible to thread 2. Thus it will go to the lower memory hierarchy searching for A and will get the value 0 which is stored in the L3 cache rather than the updated one. To solve this the concept of DRF comes into existence. All modern-day processor has some kind of store buffer. Here they take the help of a compiler which introduces fences to reduce these kinds of situations.

1.2 What are we focusing on?

CPU's with OOO processor requires few units to maintain the program order. In the case of the memory operation, the first 2 steps are common for both the store and the load. The first step is the memory address generation. The register or the main memory can be accessed by an-bit address. Unlike the registers, the main memory is not accessed using a specified address that might be stored at some location. This is because mainly the main memory is huge and thus storing an individual address for each location is not a good solution. The address is generally calculated based on a register and an offset. This takes place in the first cycle and is the same for both the store and load. In the second cycle, this address is translated to the real hardware address. Each program has a part of the main memory that it uses as a private memory. So a translation is required from this virtual address to the real physical address. This is also the same for both the memory operation and is done in the second cycle. The third cycle differs for both of them. In this work, we focus on stores only. For the store operation after the address translation, they are considered performed form the architectural point of you as they are moved to the Store queue. This is because if the speculation was wrong we need to squash and thus keeping the store in a buffer and not moving it to memory is a better choice. From the store queue, the store moves to the store buffer. In real processor, the SQ and SB are the same registers so it just shifting the stores toward the head.

In a multi-core processor, each core has its unit of store queue and store buffer for managing store operations. In a real processor, the store queue and store buffer are unified. Unified in the sense that they have the same physical register and the partition is done logically. The store queue is needed to manage the dynamic out of order executions inside the core. Once the store is committed it is moved to the store buffer. As stated above they share the same physical register. The buffer is generally a FIFO. The requests are inserted from the tail are performed then dequeue from the head making a place for other stores.

Once the buffer is full all other stores have to wait. This generates a stall and reduces performance. It gets even worse if the store at the head which is writing to the memory is a miss. A Nobel way would be to start the other stores to perform in the memory if one gets a miss. But this will change the program order.

Suppose we have the following store operations:-

Store A
Store B
Store C
Store D

At some point out store buffer will look like this :-

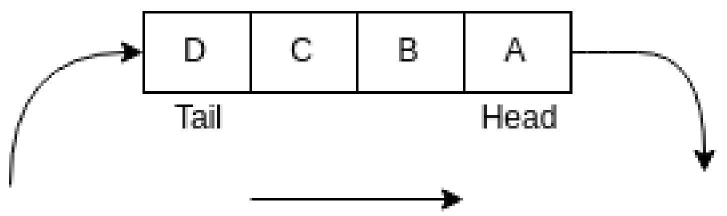


Figure 5: *Store Buffer*

According to the program order, then A will be at the head as compared to another store. Suppose A gets a miss and we start B then what we are doing is storing B then A and then C and D which is the wrong behaviors as according to programmer we should store A first then B. Now we at a point to discuss our solution. What if we can identify the stores that we can re-order which means we can perform them even if they are not at the head as we know they will not disturb the program order. This is already a field of research and we can find many solutions to this. Most of them use the word data race free or DRF which means these store does not depend on the program order.

In the parallel programming, some part of the code is visible or we can say are shared with the other codes and this region which is shared between different codes are called critical section. To simplify reasoning about the correctness of parallel executions, mainstream languages such as C++ and Java have already adopted data-race-free (DRF) as a standard and provide none or weak guarantees in the presence of data races.

The DRF confirms that this region will not be shared with the other threads. These regions are also called as synchronization-free regions. In

c++ multi-thread programming is implemented by using Mutex. Mutex provides different functions that can be used to control the part of the code you want to share with the other threads. Once you figure out the region you want to share all the variable which are been used in that regions will affect the other threads output. All the variables which are not in the part of the critical section are not shared between the threads. This allows us to apply an optimization change which is to re-order the variables which are not in the critical section and before starting the critical section we wait for all the memory operations from the non-critical sections to complete.
The image above shows the regions, the middle region is the critical section

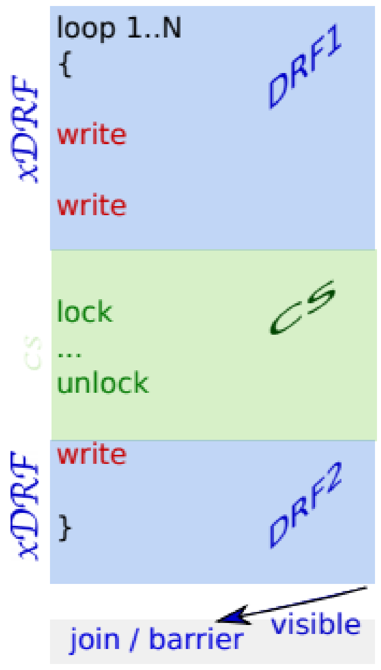


Figure 6: *xDRF region*

which is been visible to the other threads.[3]

So, in conclusion, we can re-order the stores that are in the xDRF region and other stores should be performed in the program order. Also, all the stores which belong to a DRF region should be performed before re-ordering any other store from other DRF region. This is to maintain the program order.

2 How did we generate the workload?

Our workload is a trace file that is generated using a binary file generated by real hardware. The binary file already contains all the information regarding DRF's by the compiler. But for our processor, we need to add a flag or any instruction that will tell him that this is the boundary of XDRF or NDRF. Our final workload look like the following:-

```
404e42
S0d15m129 7ffd95ad8e50 8
1d130 3 L1d132 7ffd95ad8e50 8
L1d133 7ffd95ad8e58 8
0d1 -9956 BEGIN_iNDRF 7 5d1 L2 607998 8
7 END_iNDRF 7 5d1 2d3 4d4 4d5 7d4 2d7 2 S0d1m146 7ffd95ad8e58 8
-4204429
END_XDRF 0
4214337 L1d149 7ffd95ad8e58 8
0d1 -9904d8 2d8 2 S0d1m154 7ffd95ad8e58 8
-4204438
BEGIN_NDRF 0
4214289 L1d157 7ffd95ad8e58 8
0d1 -9847d19 3d18 3d20 3 S0d1m163 7ffd95ad8e58 8
-4204452
END_XDRF 1
BEGIN_NDRF 1
END_XDRF 2
BEGIN_NDRF 2
END_XDRF 3
BEGIN_NDRF 3
END_XDRF 4
BEGIN_NDRF 4
END_XDRF 5
BEGIN_NDRF 5
END_XDRF 6
BEGIN_NDRF 6
```

```
END_XDRF 7
BEGIN_NDRF 7
END_XDRF 8
BEGIN_NDRF 8
END_XDRF 9
BEGIN_NDRF 9
END_XDRF 10
BEGIN_NDRF 10
END_XDRF 11
BEGIN_NDRF 11
.
.
.
```

We can see it contain all the information about store the address the data and also include the DRF information. To generate these traces we used Sniper tool[4]. We used Splash-3 as a benchmark for this project[5]. Splash-3 is an improved version of a very famous benchmark called Splash-2. Splash-3 provides a better and bug-free benchmark. Overall the Splash benchmark is the most popular in the field of multi-thread application. It is composed of eleven workloads, three of which come in two implementations that feature different optimizations.[6] The majority of workloads belong to the High-Performance Computing domain.

2.1 Sniper

Sniper [4] is a next-generation parallel, high-speed and accurate x86 simulator. This multi-core simulator is based on the interval core model and the Graphite simulation infrastructure, allowing for fast and accurate simulation and for trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures.

The Sniper simulator allows one to perform timing simulations for both multi-program workloads and multi-threaded, shared-memory applications with 10s to 100+ cores, at a high speed when compared to existing simulators. The main feature of the simulator is its core model which is based on interval simulation, a fast mechanistic core model. Interval simulation raises the level of abstraction in architectural simulation which allows for faster simulator development and evaluation times; it does so by 'jumping' between miss events, called intervals. Sniper has been validated against multi-socket Intel Core2 and Nehalem systems and provides average performance prediction errors within 25 percent at a simulation speed of up to several MIPS.

This simulator and the interval core model is useful for unicore and system-level studies that require more detail than the typical one-IPC models, but for which cycle-accurate simulators are too slow to allow workloads of meaningful sizes to be simulated. As an added benefit, the interval core model allows the generation of CPI stacks, which show the number of cycles lost due to different characteristics of the system, like the cache hierarchy or branch predictor, and leads to a better understanding of each component's effect on total system performance. This extends the use for Sniper to application characterization and hardware/software co-design.

The xDRF and NDRF regions are introduced in the binary by 2 things:-

- Introducing XCHG
- Introducing Function calls

The XCHG instruction is only used to mark the NDRF region while the function call is used for both the NDRF and DRF region. The XCHG instruction just exchanges the values between two registers and thus XCHG %EDI, %EDI will be an NOP instruction. The values of the register define whether it is a begin or an end of the NDRF region. EDI register with value -7 represents the end of the NDRF region while the +7 represents the begin of the NDRF region.

While the XCHG just used for NDRF the function calls are used for both the regions DRF and the NDRF. Several other functions are also inserted at a particular point like Barrier, lock acquire, etc for compiler information. The functions also return certain integer values to differentiate between several xDRF and NDRF.

2.2 Instruction Modeling in SNIPER

The sniper models instructions in two separate ways one is static and one is dynamical. The static instruction just searches for the particular instruction already modeled in the PIN tool by INTEL [7]. While the dynamic instruction contains the data like branch target etc. For finding whether an instruction is XCHG or not can be done directly in the instruction modeling file. The SNIPER uses an interface of the PIN tool provided to model the binaries. For searching whether the instruction is an XCHG instruction or not we can use the following piece of code.

```
if (INS_IsXchg(ins)) {
 assert(INS_OperandCount(ins) == 2);
 if (INS_OperandReg(ins, 0) == INS_OperandReg(ins, 1)
            && INS_OperandReg(ins, 0) == REG_EDI) {
    INSTRUMENT_PREDICATED(
    INSTR_IF_DETAILED(inst_mode),
    trace, ins, IPOINT_BEFORE, (AFUNPTR)handleXchg,
    IARG_CONST_CONTEXT,
    IARG_THREAD_ID,
    IARG_END);
     }
   }
```

The INS _IsXchg is the function provided in the PIN tool API which allows us to check whether the instruction we are modeling is XCHG or not. Once we get the XCHG instruction we were looking for we jump to a function that is handleXchg.

The function definition is given below:-

```
static void handleXchg(const CONTEXT *ctxt, THREADID thread_id) {
  REG reg = REG_EDI;
  assert(!REG_is_gr8(reg) && !REG_is_gr16(reg));
  ADDRINT regVal;
  regVal = PIN_GetContextReg(ctxt, REG_FullRegName(reg));
  if (int(regVal) > 0) {
    //cerr << thread_id << " BEGIN_iDRF " << int(regVal) << endl;
  } else {
    //cerr << thread_id << " END_iDRF " << 0 - int(regVal) << endl;
  }
  assert(localStore[thread_id].dynins);
  localStore[thread_id].dynins->setXchgRegValue(int(regVal));
}
```

Once we get the XCHG instruction we will take the register value of EDI register as this will be used to check weather its an END or BEGIN of the region. The value +7 is the begin of the region while -7 represent the end of the region. Once the value is obtained, a function which is been defined in the Dynamic Instruction class is called and the values are passed to that class. The register values should be passed dynamically just like the branch target as it can change according to the execution of the code.

Later from Dynamic Instruction, the values are passed to the DynamicMicroOp class and from DynamicMicroOp it is finally passed to the Trace Generator.

2.3 Function Calls In SNIPER

All the function call in the SNIPER are modeled in the routine replace the file in the pin/lite. Here we have to look for the function which is XDRF and NDRF and is introduced by the compiler to mark the regions. So here our task was to search for that specific function calls and then pass the information to the Trace file where we can finally print them in the trace file.

```
 else if (name.find("begin_XDRF") != string::npos){
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)sendTraceInsn, IARG_THREAD_ID,
                                             IARG_ADDRINT, 19,
                                             IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                             IARG_ADDRINT, 0,
                                             IARG_ADDRINT, 0,
                                             IARG_END);
  } else if (name.find("end_XDRF") != string::npos){
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)sendTraceInsn, IARG_THREAD_ID,
                                              IARG_ADDRINT, 20,
                                              IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_END);
  } else if (name.find("end_NDRF_BARRIER") != string::npos){
   RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)sendTraceInsn, IARG_THREAD_ID,
                                              IARG_ADDRINT, 21,
                                              IARG_ADDRINT, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_END);
  } else if (name.find("begin_NDRF") != string::npos) {
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)sendTraceInsn, IARG_THREAD_ID,
                                              IARG_ADDRINT, 22,
                                              IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_END);
  } else if (name.find("end_NDRF") != string::npos){
    RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)sendTraceInsn, IARG_THREAD_ID,
                                              IARG_ADDRINT, 23,
                                              IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_END);

  } else if (name.compare("pthread_join") == 0
        || name.compare("pthread_mutex_lock") == 0
        || name.compare("__pthread_mutex_lock") == 0
        || name.compare("pthread_mutex_unlock") == 0
        || name.compare("__pthread_mutex_unlock") == 0
        || name.compare("pthread_cond_signal") == 0
        || name.compare("pthread_cond_broadcast") == 0
        || name.compare("pthread_cond_wait") == 0
        || name.compare("sem_post") == 0
        || name.compare("sem_wait") == 0){
  RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)sendTraceInsn, IARG_THREAD_ID,
                                              IARG_ADDRINT, 24,
                                              IARG_ADDRINT,0,
                                              IARG_ADDRINT, 0,
                                              IARG_ADDRINT, 0,
                                              IARG_END);
```

Once we get our function we call a function which is sendTraceInsn which passes all the values to the current MicroOp class. The RTN InsertCall is a function provided by the PIN tool which inserts a call between the instruction and thus allows us to do some specific tasks. The arguments are explained below:-

- rent is the routine to the instrument which means the routine we want to examine.
- IPOINT_BEFORE This means to call the function which is AFUNPTR before.
- (AFUNPTR)sendTraceInsn This is the function that is being called.
- IARG_THREAD_ID This passes the current thread
- IARG_FUNCARG_ENTRYPOINT_VALUE This passes the value which is being passed by the function call.
- IARG_ADDRINT This is a simple int argument we can pass any value using this. In our case, we pass different values for different cases. For example, for Begin_XDRF we pass 19 and for the end, we pass 20 and so on.
- IARG_END This means that no more arguments will be passed.

The definition of the function called by the above are given below:-

```
VOID sendTraceInsn(THREADED id, ADDRINT type, ADDRINT arg0,
ADDRINT arg1, ADDRINT arg2)
{
   //std::cerr << __FUNCTION__ << " called! type=" << type << "\n";

   OperandList list;
   Instruction *inst = new GenericInstruction(list);
   inst->setAddress(0xffffffffffffffff);
   inst->setSize(15);
   inst->setDisassembly("RMS");
   MicroOp *currentMicroOp = new MicroOp();
   currentMicroOp->makeExecute(0 /*offset*/, 0 /*num_loads*/, XED_ICLASS_NOP
                         /*instructionOpcode*/, "RMS", false /*isBranch*/);
   currentMicroOp->setInstruction(inst);
   currentMicroOp->setFirst(true);
   currentMicroOp->setLast(true);
   currentMicroOp->is_trace = true;
   currentMicroOp->trace_data[0] = type;
   currentMicroOp->trace_data[1] = arg0;
   currentMicroOp->trace_data[2] = arg1;
   currentMicroOp->trace_data[3] = arg2;
   std::vector<const MicroOp *> * uops = new std::vector<const MicroOp*>();
   uops->push_back(currentMicroOp);
   inst->setMicroOps(uops);
   InstructionModeling::handleInstruction(id, inst);
}
```

In the definition, we save the value passed by the arguments in type, arg0, arg1, arg2. Later these are passed to the current micro-op using:-

```
   currentMicroOp->trace_data[0] = type;
   currentMicroOp->trace_data[1] = arg0;
   currentMicroOp->trace_data[2] = arg1;
   currentMicroOp->trace_data[3] = arg2;
```

Thus till this stage, we have passed all the information we needed to the trace file. Now we need to change the trace file and print the values we want.

2.4 Trace Configuration

The trace is generated in the rob_timer.cc in the rob_trace folder. The trace contains all the information about the instruction fetched and there address, dependencies and the size of the instruction. A sample of the trace is given below:-

```
4214321 L1d219 7ffd95ad8e58 8
0d1 L-9146 607984 4
6d1 4d1 3d1 4 S0d1m226 7ffd95ad8e58 8
L-7768 607098 8
0d1 S139885341901344m229 7ffd95ad8e48 8
S5d9m230 7ffd95ad8e38 8
5d7 S3d10m232 7ffd95ad8e40 8
S5m233 7ffd95ad8e50 8
5d234 L4 7f3998908ea0 8
L7d1 7f3998909700 8
3d1 b3d1t490 L6 7f3998908d20 8
L7d1 7f39984626c0 8
4d1 b3d1t340 6 5d5 L2 7f399890f1b4 4
0d1 b7d1t14 L2d8 7f3998909720 4
0d1d5d6 S0d1m10 7f3998909720 4
b4d2t8924 6 11d13 3d23 3d2 3 S0d1m23 7ffd95ad8e28 8
S-11504m24 7ffd95ad8e20 8
S2m25 7ffd95ad8e18 8
S2m26 7ffd95ad8e10 8
2d7 S3d31m28 7ffd95ad8e08 8
S2d41m29 7ffd95ad8e00 8
S1d11m30 7ffd95ad8df8 8
```

For example, let's take the 1st line L1d219 7ffd95ad8e58 8, L and S represent the load and store operation. L1d219 represents that the load is dependent on the instruction at 1d219 memory ahead of the current instruction pointer. 7ffd95ad8e58 represents the address at which the load or store operation is performed. While at the last the 8 represents the size of the instruction. The trace can be generated over several threads and contain the traces of individual threads.

For tracing the instructions we used the following code :-

```
if (dmo.isXchg()) {
         if (dmo.getRegValue() > 0)
            deptrace_f << "BEGIN_iNDRF " << dmo.getRegValue() << " ";
         else
            deptrace_f << "END_iNDRF " << 0 - dmo.getRegValue() << " ";
          }
```

We call the function from the Dynamic MicroOp class which tells us to weather the instruction is the XCHG instruction we are looking for if its true we take the register value and print accordingly the end and begin of the NDRF region.

For tracing the function call:-

```
case 19:
        if (deptraceRMSIsActive(thread_id))
        {
           if (deptrace_last_was_newline)
               deptrace_last_was_newline = false;
            else
               deptrace_f << "\n";

   deptrace_f << "BEGIN_XDRF " << (*it)->getMicroOp()->trace_data[1] << "\n";
   deptrace_last_was_newline = true;
        }
      break;
        // BEGIN_XDRF
        case 20:
           if (deptraceRMSIsActive(thread_id))
           {
              if (deptrace_last_was_newline)
                 deptrace_last_was_newline = false;
              else
                 deptrace_f << "\n";

     deptrace_f << "END_XDRF " << (*it)->getMicroOp()->trace_data[1] << "\n";
     deptrace_last_was_newline = true;
       }
           break;
```

```
    // BEGIN_NDRF_BARRIER
     case 21:
   assert(false); // Not tested
    break;
      // BEGIN_N92.2DRF
     case 22:
        if (deptraceRMSIsActive(thread_id))
        {
           if (deptrace_last_was_newline)
              deptrace_last_was_newline = false;
           else
              deptrace_f << "\n";

    deptrace_f << "BEmcGIN_NDRF " << (*it)->getMicroOp()->trace_data[1]
    << "\n";
    deptrace_last_was_newline = true;
    }
        break;
     // END_NDRF
     case 23:
        if (deptraceRMSImcsActive(thread_id))
        {
           if (deptrace_last_was_newline)
              deptrace_last_was_newline = false;
           else
              deptrace_f << "\n";

     deptrace_f << "END_NDRF " << (*it)->getMicroOp()->trace_data[1]
     << "\n";
     deptrace_last_was_newline = true;
    }
        break;
```

We use the switch(type) at the starting and as shown in the previous section we send some particular value fro different function calls. Then we simply print our data. The deptraceRMSActive() check whether the printing in the trace is allowed on not this is done as some of the functions are printed at there END calls and at there begin to call only the values passed by the functions are stored. mc

After this, the SNIPER can generate traces by checking the XDRF and the NDRF regions which we inserted using our compiler.

3 How does our processor identify DRF regions?

Every execution starts by reading the instruction and then processing them and making memory operation if required. Generally, these instructions in collective are called Instruction Set that defines a bunch of Instructions generally used to program any kind of code. Each Instruction has its hexadecimal representation that is what we called machine code. The compiler does the work to transfer the code from human-readable form to machine code.

Reading the instructions in our simulator is done is file TraceRecordPThreads.C in the folder recorder. Each instruction is modeled by a user-defined data type. All the operations are declared in the file OperationType.h in the system folder.

```
enum OperationType {
  OperationType_None,
  OperationType_FIRST = OperationType_None,
  OperationType_Instruction,
  OperationType_Memory,
  OperationType_Branch,
  OperationType_Memory_Func,
  OperationType_Branch_Func,
  OperationType_Clear_Stats,
  OperationType_Lock_Acquire,
  OperationType_Lock_Release,
  OperationType_Barrier,
  OperationType_Cond_Signal,
  OperationType_Cond_Broadcast,
  OperationType_Cond_Wait,mc
  OperationType_Sem_Signal,
  OperationType_Sem_Wait,
  OperationType_Spin,
  OperationType_Fence,
  OperationType_xDRF_Begin,
  OperationType_xDRF_End,
  OperationType_nDRF_Begin,
  OperationType_nDRF_End,
  OperationType_Parallel_Begin,
  OperationType_Parallel_End,
  OperationType_NUM
};
```

This shows all the operation types we have in our simulator. You can notice that we have OperationType_xDRF_Begin, OperationType_xDRF_End, OperationType_nDRF_Begin, OperationType_nDRF_End and not OperationType_xDRF as there is no operation like XDRF, XDRF, and NDRF and just a representation of the region where we can reorder and regions where we can not reorder. That is why we decided to put the operation type with the end and begin that represent the regions. All the instructions falling in between OperationType_xDRF_Begin and OperationType_xDRF_End are in the region where we can allow reordering of instructions. In our case, we are focusing on store operations. Once we know that the operation is of XDRF_begin we can pass that information to the processor by activating a special flag. All stores then copy this bit with them that help processor identify weather the sotres are DRF or NDRF(shown in figure ??). In our simulator we call the following functions:-

```
m_proc_ptr->setndrf(true);
m_proc_ptr->setxdrf(false);
```

These two functions are defined to set the regions to true or false according to the instruction read from the trace.

Alongside, this instruction we also use different fencing that helps us to define DRF regions these fences are generally put by the compiler. One such example is Lock, so when we see an operation with lock acquire all the operations after that is NDRF until we find the release for the lock. Another fencing is a barrier. The implementation is provided below:-

```
else if (m_current_func.operation == OperationType_Lock_Acquire) {
   assert(isAccessAligned(m_current_func.data_address, SYNC_VAR_SIZE));
   m_request_finished = m_lock_ptr->acquire(m_current_pc,
       m_current_func.data_address);
   if (FENCING_STRATEGY == 7){
     if(m_proc_ptr->isndrf() == false){
   if (count_drf == 0){
     m_proc_ptr->setndrf(true);
     m_proc_ptr->setxdrf(false);
```

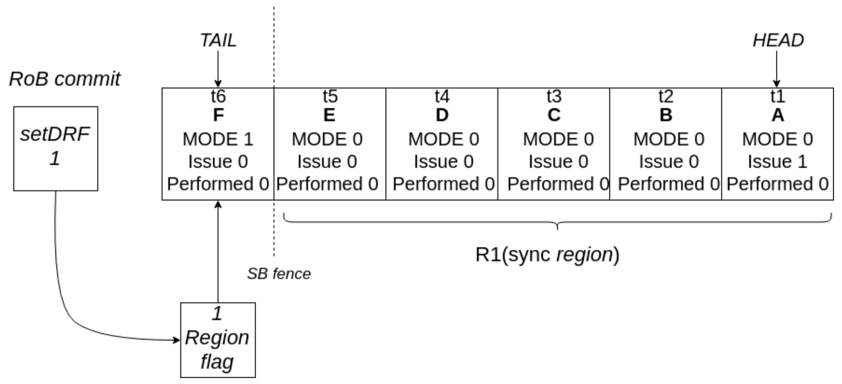

TAIL
HEAD
RoB commit
setDRF 1
t6 F MODE 1 Issue 0 Performed 0
t5 E MODE 0 Issue 0 Performed 0
t4 D MODE 0 Issue 0 Performed 0
t3 C MODE 0 Issue 0 Performed 0
t2 B MODE 0 Issue 0 Performed 0
t1 A MODE 0 Issue 1 Performed 0
R1(sync region)
SB fence
1 Region flag

Figure 7: *By default, stores are considered unsafe to reorder, until a OperationType_xDRF_Begin instruction is executed. Stores A, B, C, D, and E copy the region flag 0 and thus belong to a sync region (Mode bit 0). Once a OperationType_xDRF_Begin operation reaches the RoB commit head, the processor sets the region flag and inserts a logical SB fence, marking the beginning of a DRF region. Store F that enters after OperationType_xDRF_Begin copies in its Mode bit the region flag's value, which is now 1.*

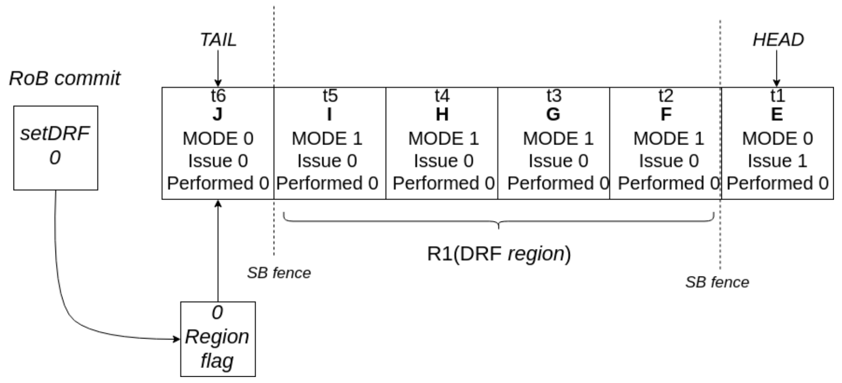

TAIL
HEAD
RoB commit
setDRF 0
t6 J MODE 0 Issue 0 Performed 0
t5 I MODE 1 Issue 0 Performed 0
t4 H MODE 1 Issue 0 Performed 0
t3 G MODE 1 Issue 0 Performed 0
t2 F MODE 1 Issue 0 Performed 0
t1 E MODE 0 Issue 1 Performed 0
R1(DRF region)
SB fence
SB fence
0 Region flag

Figure 8: *Operation OperationType_xDRF_End marks the end of a DRF region: it resets the region flag and triggers the insertion of an SB fence. As seen before, store J copies the current value of the region flag in its Mode bit. Value 0 indicates it is a sync store. (Stores F, G, H and I copied the value of the region flag at the moment the stores entered the SB, marking them as DRF.)*

```
    }
      }
      if (m_request_finished)
    count_drf++;
    }
  } else if (m_current_func.operation == OperationType_Lock_Release) {
    assert(isAccessAligned(m_current_func.data_address, SYNC_VAR_SIZE));
    m_request_finished = m_lock_ptr->release(m_current_pc,
    m_current_func.data_address);
    if (FENCING_STRATEGY == 7){
      if(m_request_finished) count_drf--;
      if(count_drf == 0){
    assert(m_proc_ptr->isndrf() == true);
    m_proc_ptr->setndrf(false);
    m_proc_ptr->setxdrf(true);
      }
      assert(count_drf >= 0);
    }
  } else if (m_current_func.operatmcion == OperationType_Barrier) {
    assert(isAccessAligned(m_current_func.cond_address, SYNC_VAR_SIZE));
    assert(isAccessAligned(m_current_func.count_address, SYNC_VAR_SIZE));
    assert(isAccessAligned(m_current_func.lock_address, SYNC_VAR_SIZE));
    m_request_finished = m_barrier_ptr->barrier(m_current_pc,
                  m_current_func.cond_address,
                  m_current_func.count_address,
                  m_current_func.lock_address);

    if (FENCING_STRATEGY == 7){
      if (m_proc_ptr->isndrf() == false){
    assert(count_drf == 0);
    m_proc_ptr->setndrf(true);mc
    m_proc_ptr->setxdrf(false);
      }//begin
      if (m_request_finished) {
    m_proc_ptr->setndrf(false);
    m_proc_ptr->setxdrf(true);
      } //end
    }
```

One confusing word at this stage is FENCING_STRATEGY but that is what we are discussing in the next section.

4 How does these DRF are defined?

Overall there are 2 ways to define these regions one is doing it manually the other option is to take the help of compiler to add this information. In our system, we implement several different strategies to identify these regions. The manual is done by putting fencing while the compiler work is done based on this paper [3].

All modern-day programming languages proved DRF support[8]. But passing that information to the processor is not that easy. A solution is to track loads and stores per region. But this has two key issues. Store order and load-to-store forwarding. That is reorderings to the same variable lead to incorrect program behavior for example if we have 2 requests of the store to the same address according to the programmer the value in the memory should be the of the store which is near to the tail in the store buffer. But if we do reordering we may end up performing the wrong store request. But this can be easily solved by inserting fences that do not allow any store to perform until all the store before that fence is performed in the memory.

The other issue of load-to-store forwarding arises as if we have two requests of the same address and we reorder then and thus one is performed but when the load will scoop the store buffer for data it will take the data from the other store of the same address thus the results will not be valid anymore, but if we are sure that the reordering only happens when we have an exclusive access to the particular memory address that means no other requests such load or store will access that memory we can solve this issue. As we only recorder in DRF regions thus we are sure there will be no load trying to access the same memory location thus keeping the results accurate and still giving some opportunities to optimize.

We exploit this with a simple technique, by adding fences (xDRF fences) in reordering phases. Thus getting a lot of space to reorder things thus hiding the timing of cache miss. Our observation is that compilers can easily detect this (potential violation) and then insert fences. These fences can easily be used to pass the DRF information to the store buffer. Once this information is passed to the store buffer the memory model of the processor can be relaxed to a great extent. Relaxing the memory model helps us reduce the stalls and hide the timing of writing to the cache of a xDRF store.

In our system following are the different strategies. All are defined in a ruby default file in the config folder.

In our traces, all the fencing are already implemented and we can choose which fencing strategy to use every time using our config file. The type of fencing is represented by the number as we discussed earlier. For a quick example consider BEGIN_NDRF 10 where the value 10 determines the fencing strategy. In our system, we check for the fencing strategy with all the fencing strategies from the traces.

```
else if (type.compare("BEGIN_XDRF") == 0) {
   m_current_func.operation = OperationType_xDRF_Begin;
   int value_fence;
   in >> value_fence;
   assert(value_fence <= 12);
   m_current_func.size = getXDRFVersion(value_fence);mc
```

As we can see above once we match the BEGIN_XDRF we set the operation type as begin and we put the value of the integer at value_fence. Then the getXDRFVersion function is called which match the number with the list of fencing strategies. The code is shown below:-

```
int TraceRecordPThreads::getXDRFVersion(int value) const {
 switch (value) {
 case 0:
   return FENCING_STRATEGY_xDRF_MANUAL;
 case 1:
   return FENCING_STRATEGY_xDRF_LLVM_MAYALIAS;
 case 2:
   return FENCING_STRATEGY_xDRF_SVF_MAYALIAS;
 case 3:
   return FENCING_STRATEGY_xDRF_USECHAIN_MAYALIAS;
 case 4:
   return FENCING_STRATEGY_xDRF_LLVM_MUSTALIAS;
 case 5:
   return FENCING_STRATEGY_xDRF_SVF_MUSTALIAS;
 case 6:
   return FENCING_STRATEGY_xDRF_USECHAIN_MUSTALIAS;
 case 7:
   return FENCING_STRATEGY_xDRF_LLVM_MAYALIAS_CONF;
 case 8:
   return FENCING_STRATEGY_xDRF_SVF_MAYALIAS_CONF;
 case 9:
   return FENCING_STRATEGY_xDRF_USECHAIN_MAYALIAS_CONF;
```

```
  case 10:
    return FENCING_STRATEGY_xDRF_LLVM_MUSTALIAS_CONF;
  case 11:
    return FENCING_STRATEGY_xDRF_SVF_MUSTALIAS_CONF;
  case 12:
    return FENCING_STRATEGY_xDRF_USECHAIN_MUSTALIAS_CONF;
  default:
    cout<<"Fencing value "<<value<<"\n";
    ERROR_MSG("Invalid range for type xDRF\n");
    return -1;
  }
```

Once we have the fencing strategy from the trace we compare it with the fencing strategy we set in our simulator and only begin the DRF regions when it matches.

```
 else if (m_current_func.operation == OperationType_xDRF_Begin) {
    if (FENCING_STRATEGY == m_current_func.size){
      m_proc_ptr->setxdrf(true);
      m_proc_ptr->set_begin_xdrf();
```

The manual DRF which is being represented by fencing strategy number 7 is done without the help of compiler and thus the fencing that is present in the trace is used to define the DRF regions. These are done as follows:-

```
 if (FENCING_STRATEGY == 7){
      if (m_proc_ptr->isndrf() == false){
    assert(count_drf == 0);
    m_proc_ptr->setndrf(true);
    m_proc_ptr->setxdrf(false);
      }//begin
      if (m_request_finished) {
    m_proc_ptr->setndrf(false);
    m_proc_ptr->setxdrf(true);
      } //end
    }
```

So at the start of the fence, we set the NDRF to true and the XDRF to false, which means all the instructions from this point will be considered at NDRF and thus no store will re-ordered. Once the fence is ended we set the NDRF back to false and the XDRF back to true.

In the result section, you can see the graphs that show the number of stores committed in the XDRF regions and as predicted the compiler does a much better job as compared to manual.

5 How do we process stores in different DRF regions?

Once the processor has the information about the regions we created a new type of store buffer which we called Hybrid store buffer as it can re-order and also can order things in order depending on the region.

All the stores once committed and put in the store buffer where they wait to perform finally in the memory. A function called consumeCycle is called that tries to write the stores in the memory. The function can be seen below:-

```
void HybridStoreBuffer::consumeCycle(bool atomic_decoded) {
 int number_of_writes = 0;
 head_xdrf = m_store_buffer_ptr->peekHead().m_xdrf;
 if (m_issue_policy == ASAP_head
     || (m_issue_policy == Delay_Sentinel
     && m_store_buffer_ptr->peekHead().m_Sentinel == 0)) {
   for (int i = m_store_buffer_ptr->fromHeadBegin();
   m_store_buffer_ptr->more();
    i = m_store_buffer_ptr->fromHeadNext(i)) {
     if(m_store_buffer_ptr->getEntry(i).m_xdrf != head_xdrf){
   g_system_ptr->getProfiler()->drffence();
     }
     if(m_store_buffer_ptr->getEntry(i).m_xdrf != head_xdrf
     && m_sequencer_ptr->isStoreFree())break;
     if(m_store_buffer_ptr->getEntry(i).m_ndrf == false
     && m_store_buffer_ptr->getEntry(i).m_Locked == false){
     //can be reordered
   //print(cerr);
   if(tryToBeginWriteToCache(m_store_buffer_ptr->getEntry(i))){
     number_of_writes++;
     break;
   }
```

```
      }
      if(m_store_buffer_ptr->getEntry(i).m_ndrf == true
      && m_store_buffer_ptr->getEntry(i).m_Locked == false
      && m_pending == false){ //can not be reordered
    //print(cerr);
    if(tryToBeginWriteToCache(m_store_buffer_ptr->getEntry(i))){
      number_of_writes++;
      }
    else
      break;
      }
    }
  }
  while(!isEmpty() && m_store_buffer_ptr->peekHead().m_Locked
    && m_store_buffer_ptr->peekHead().m_IsPerformed) {
    m_store_buffer_ptr->dequeue(); // remove from head
  }
  assert(number_of_writes <= CACHE_WRITE_PORTS);
  // Try to prefetch
  for (int i = m_store_buffer_ptr->fromHeadBegin(); m_store_buffer_ptr->more();
       i = m_store_buffer_ptr->fromHeadNext(i)) { // From head to tail
    StoreBufferEntry& entry = m_store_buffer_ptr->getEntry(i);
    if (entry.m_TryPrefetch) {
      if(tryToPrefetch(entry)){
      return; // One prefetch per cycle
      }
    }
  }
  return;
}
```

Once a function is called we look for each store from Head to Tail and the oldest request will be at the head and the newest will be towards the tail. Also, we always dequeue from the head so we always start from the head and try to write the request at the head to the memory.

```
for (int i = m_store_buffer_ptr->fromHeadBegin(); m_store_buffer_ptr->more();
    i = m_store_buffer_ptr->fromHeadNext(i)) {
```

Each store already has information about the DRF. The DRF information of whether the store was in the NDRF region or XDRF region is attached to them when they are pushed into the pipeline.

```
PipelineElement(const Instruction &i, bool n_flag,
bool x_flag, bool atomic, bool m_begin_xdrf,
bool m_end_xdrf) : instr(i) {
    stage = PipelineStage_FETCH;
    // phy_addr initialized later
    prefetch_issued = false; non_prefetch_issued = false;
    Dspeculatively_executed = false; Mspeculatively_executed = false;
    is_Dspeculative = false; is_Mspeculative = false; mem_resolved = false;
    conflict_store_positions.clear();
    needs_reexec = false; seen_write = false;
    committed_OoO = false; has_lockdown_table_entry = false;
    locks_cache_entry = false; locks_sb_entry = false;
    rob_position = 0; sentinel = 0; clear_sentinels = false;
    sync_type.clear(); ndrf_bit = n_flag; xdrf_bit = x_flag;
    atomic_bit = atomic; begin_xdrf = m_begin_xdrf; end_xdrf = m_end_xdrf;
 }
```

Before issuing any store we check for the DRF boundary that means we wait until the specific DRF that is being executed is finished as if we don't wait we will break the consistency rule. For this, we just check the changes in the DRF bit. So if the DRF bit is 1 1 1 1 that means they all belong to the region where we can reorder then while if the bits are something like 1 1 1 1 0 0 that means at the change from 1 to 0 we have to wait until all the requests having DRF bit 1 finish their process of writing the data to the memory. To ensure this we have to do 2 steps:-

- Don't allow any store request in store buffer to try writing to memory.
- Wait until all the ongoing request to write into the memory is finished.

For the first one we just take the Head's DRF bit and check it with all the upcoming requests if they are the same we allow them to process the operation based on there DRF bit but if they are not equal we break the loop and thus does not allow them to process any more request until they are done.

```
if(m_store_buffer_ptr->getEntry(i).m_xdrf != head_xdrf &&
m_sequencer_ptr->isStoreFree())break;
```

The function instore free tells us whether there are any pending or on-going memory request or not. It is implemented in the Sequencer and the implementation is given below:-

```
bool Sequencer::isStoreFree(){
  int total_demand = 0;
  Vector<Address> keys = m_readRequestTable_ptr->keys();
  keys = m_writeRequestTable_ptr->keys();
  for (int i=0; i< keys.size(); i++) {
    CacheMsg& request = m_writeRequestTable_ptr->lookup(keys[i]);
    if (!isPrefetch(request)) {
      total_demand++;
    }
  }

  if(m_l0controller.countActualWrites() == 0 && total_demand == 0){
    return true;}
  else{
    return false;}
```

Once we are sure we can allow stores to start trying to write to the memory we just check the DRF bit and allow them to try either in order or in out of order.

```
   if(m_store_buffer_ptr->getEntry(i).m_ndrf == false
   && m_store_buffer_ptr->getEntry(i).m_Locked == false){
   //can be reordered
    //print(cerr);
    if(tryToBeginWriteToCache(m_store_buffer_ptr->getEntry(i))){
      number_of_writes++;
      break;
    }
      }
```

The DRF bit we discussed before is m_NDRF so if it is true that means that store belongs to the NDRF region and thus can not be re-ordered but if its false that means its DRF as all stores except m_NDRF false are DRF and are called XDRF(Extended Data Race Free). So once we see that the store has a m_ndrf bit false and the locked bit is false as well we can allow them to start writing to the memory. The locked bit tells us whether the store is being already issued or not. So we only issue the stores that are not issued yet. Once these both conditions are satisfied we try to write by calling the function tryToBeginWriteToCache if its a success it will return true and we will increase the number of writes. Then we move to the other store and try to start writing to memory. Suppose due to some reason tryToBeginWriteToCache function return false then also we will break the $if(tryToBeginWriteToCache(m\_store\_buffer\_ptr- > getEntry(i)))$ and try to execute the next store thus reordering them.

While if you look at the code below you will find that once the tryToBeginWriteToCache returns false it will break and will try to write the same request again until it returns true thus saving the order of the requests.

```
  if(m_store_buffer_ptr->getEntry(i).m_ndrf == true
  && m_store_buffer_ptr->getEntry(i).m_Locked == false
  && m_pending == false){ //can not be reordered
  //print(cerr);
  if(tryToBeginWriteToCache(m_store_buffer_ptr->getEntry(i))){
    number_of_writes++;
    }
  else
    break;
    }
```

After discussing how we issue the requests the next thing is to remove them one by one once they are done. For this, the callback function is called.

```
  void HybridStoreBuffer::callback(const Address& addr) {
int entry_pos = markAsPerformed(addr);
assert(line_address(addr) == addr);
assert(m_store_buffer_ptr->getEntry(entry_pos).m_Address == addr);
assert(!isEmpty());
endWriteToCache(addr);
if(m_store_buffer_ptr->getEntry(entry_pos).m_ndrf == true){
  m_pending = false;
}
if (!isEmpty() && m_store_buffer_ptr->peekHead().m_Locked == true
    && m_store_buffer_ptr->peekHead().m_IsPerformed == true){
  m_store_buffer_ptr->dequeue();
}
}
```

When this function is called with the address of the request we can be sure that the request is completed and we can remove them from the store buffer. So the first thing we do is to mark the request as performed. $intentry\_pos = markAsPerformed(addr)$; function checks all the requests and match it with the address and return the positing of the store.

```
  int HybridStoreBuffer::markAsPerformed(const Address& addr) {
 for (int i = m_store_buffer_ptr->fromHeadBegin(); m_store_buffer_ptr->more();
      icommited = m_store_buffer_ptr->fromHeadNext(i)) { // From head to tail
   if (m_store_buffer_ptr->getEntry(i).m_Address == addr
   && m_store_buffer_ptr->getEntry(i).m_Locked == true
   && m_store_buffer_ptr->getEntry(i).m_IsPerformed == false) {
     m_store_buffer_ptr->getEntry(i).m_IsPerformed = true;
     // cout<<" ADDRESS markasperform" << addr <<" " <<m_node_num<<endl;
     return i;
   }
 }
 //cout<<" ADDRESS markasperform" << addr <<" " <<m_node_num<<endl;
 assert(false); // There should be at least one match
}
```

All the requests which are performed and locked are removed from the head one by one. We can also remove them many at once but that will need extra hardware and thus removing one by one make more sense.

```
  if (!isEmpty() && m_store_buffer_ptr->peekHead().m_Locked == true
     && m_store_buffer_ptr->peekHead().m_IsPerformed == true){
   m_store_buffer_ptr->dequeue();
 }
```

Now, let's discuss what happens when we have a miss. So when an XDRF store misses we don't need to wait until it tries again and we can just remove it as XDRF are safe to perform out of order and they will be performed after some time in case of miss and thus we can continue our operation without waiting for XDRF miss to resolve. Thus in sequence, if the store operation is XDRF we mark it performs and then remove it from the store buffer.

While in the case of NDRF store the case is little different as in case of miss we have to wait until its resolved completely as they are not safe to perform out of order thus at each miss a function is called from the Sequencer that change the locked bit from true to false of the next request from the head and thus all the request will be issued again. This will keep on repeating until we get a hit.

```
   void HybridStoreBuffer::informNDRFRequestMissed(const Address& addr) {
  stringstream ss;
  ss << "\n" << setw(7) << g_eventQueue_ptr->getTime()
  << setw(4) << m_node_num << setw(60)
  << "Write miss / stores canceled"
     << setw(16) << addr;
  DEBUG_MSG(STOREBUFFER_COMP, MedPrio, ss.str());
  assert(!isEmpty());
  bool NDRF_store_detected = false;
  for (int i = m_store_buffer_ptr->fromHeadBegin();
  m_storecommited_buffer_ptr->more();
       i = m_store_buffer_ptr->fromHeadNext(i)) {
    if (m_store_buffer_ptr->getEntry(i).m_ndrf && NDRF_store_detected &&
    !m_store_buffer_ptr->getEntry(i).m_IsPerformed){
      m_store_buffer_ptr->getEntry(i).m_Locked = false;
      //m_store_buffer_ptr->getEntry(i).m_IsPerformed = false;
    }
    else if(m_store_buffer_ptr->getEntry(i).m_ndrf && !NDRF_store_detected &&
        !m_store_buffer_ptr->getEntry(i).m_IsPerformed){
      NDRF_store_detected = true;
      //cout<<"address "<<m_store_buffer_ptr->getEntry(i).m_Address<<endl;
      assert(m_store_buffer_ptr->getEntry(i).m_Address == addr);
    }
  }
  m_pending = true;
  // print(cerr);
}commited
```

I the code above we can notice that the first time the function is called we don't do anything except changing *NDRF_store_detected* variable value from false to true. While the next time, it will change all the requests which are NDRF and having locked bit true to locked bit false. Thus forcing them to issue again and thus maintaining the consistency.

In the sequence at the time of miss, we delete all the entries with NDRF bit as true from the store buffer as they should try again. While in the case of XDRF request we keep them in the store buffer.

```
else if (m_store_buffer_ptr->enforcesStoreStoreOrderOnlyOnVersion()) {
 if(request.getVersion()){
   m_l0commitedcontroller.removeNDRFActualWritesExceptFirst();
   // Delete only NDRF entries.
   m_store_buffer_ptr->informNDRFRequestMissed(request.getAddress());
   // Unlock the NDRF entries
 }
   }
```

We already discussed the informNDRFRequestMissed function. The removeNDRFActualWritesExceptFirst function deletes all the NDRF entries except the first one as if its a miss all the NDRF before that store should wait until it gets a hit.

```
void removeNDRFActualWritesExceptFirst() {
      CacheMsg request = *(m_pending.begin());
      //assert(request.getVersion() == 1);
      m_pending.erase(remove_if(m_pending.begin(commited) + 1, m_pending.end(),
                [](const CacheMsg& request) {
                  return request.getType() == CacheRequestType_ST
                      && request.getPrefetch() == PrefetchBit_No
                      && request.getVersion() == 1;
                }), m_pending.end());
 };
```

So you can see we start from the second request and check all the parameters like the request type which should be store the prefetch bit and the version, the variable we used to pass the NDRF information from the pipeline to the request class. So getVersion() = 1 means we delete the committed requests who are NDRF bit is true.

This completes the main modifications we made. Although there are many other small changes we made to make the overall system more efficient and some for other tests. We add all the DRF data to the profiler so that it can be printed in the stats file. For this, we define a new function which is called to count the DRF related stats.

```
void addNDRF() { m_NDRF++; }
void addXDRF() { m_XDRF++; }
void addLDNDRF() { m_LDNDRF++; }
void addSTNDRF() { m_STNDRF++; }
void addATNDRF() { m_ATNDRF++; }
void addLDXDRF() { m_LDXDRF++; }
void addSTXcommitedDRF() { m_STXDRF++; }
void addATXcommitedDRF() { m_ATXDRF++; }
void addMNDRF() { m_MNDRF++; }
void addMXDRF() { m_MXDRF++; }
```

While for dumping the stats to the output file.

```
 out << "Number Of Instruction Fetched with NDRF region: "
 << m_NDRF << endl;
   out << "Number Of Instruction Fetched with XDRF bit region: "
   << m_XDRF << endl;
   out << "Number Of Load Instructions commited with NDRF region: "
   << m_LDNDRF << endl;
   out << "Number Of Store Instruction commited with NDRF region: "
   << m_STNDRF << endl;
   out << "Number Of Atomic Instruction commited with NDRF region: "
   << m_ATNDRF << endl;
   out << "Number Of Load Instruction commited with XDRF region: "
   << m_LDXDRF << endl;
   out << "Number Of Store Instruction commited with XDRF region: "
   << m_STXDRF << endl;
   out << "Number Of Atomic Instruction commited with XDRF region: "
   << m_ATXDRF << endl;
   out << "Number Of Memory Operation commited with NDRF region: "
   << m_MNDRF << endl;
   out << "Number Of Memory Operation commited with XDRF region: "
   << m_MXDRF << endl;
   out << "Number Of DRF fences: " << m_drffence << endl;
```

6 System Specifications

We used a modified version of the cycle-accurate GEMS simulator [9] for multi-core systems. We simulate a multi-core processor providing TSO consistency and consisting of 8 out-of-order cores. Our processor mimics an

Intel Skylake micro-architecture employing macro-op and micro-op fusion. The L1 caches are fully pipelined L1 caches and employ a stride prefetcher of degree 3. Our processor employs a single circular queue for both the store queue and Store Buffer that utilizes better the resources. GARNET [10] is used to model the interconnect. The most relevant system characteristics are displayed in Table 1.

We run all applications from the Splash-3 [5] parallel benchmark suite which is a data-race-free version of the original Splash-2 [6] benchmark suite. Splash-3 benchmark suite also includes a variety of programs with different access patterns.

Table 1: *System parameters*

Processor	
Processor Model	Intel Skylake
Fetch Width	5 instructions
Issue Width	8 ports
Allocation Queue	97 entries
Reorder Buffer	224 entries
Load Queue	72 entries
Store Queue + Store Buffer	56 entries
Memory	
Private L1 I&D caches	32KB, 8 ways, 4 hit cycles, pipelined
Private L2 cache	256KB, 8 ways, 12 hit cycles
Shared L3 cache	1MB per bank, 8 ways, 35 hit cycles
Directory	8 ways, 200% coverage of L2
Memory access time	160 cycles
Network Topology	2D Mesh

The full configuration file in attached in the appendix A.

7 Results

Figure 9 shows that the number of stores executed in DRF regions are significant. In what follows, we analyze the performance benefits of performing the DRF stores out-of-order, the impact on the time the processor stalls, and the performance improvements of employing the SB as a cache. Moreover, we report the outcome of a study on the performance sensitivity with

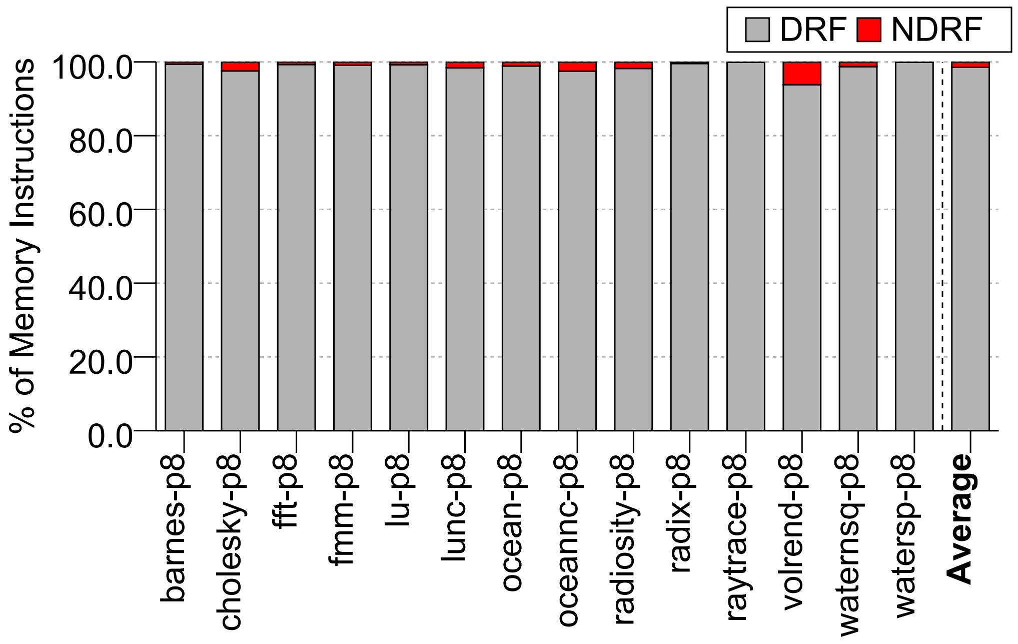


Figure 9: *Memory operations in SPLASH-3.*

respect to the SB size, and finally the impact of inserting the logical xDRF fences on every transition between regions.

7.1 Execution time

Processor stalls degrade performance considerably and are therefore an important target for speeding up applications. The more stalls the applications encounter running on the baseline SB (TSO, 56 SB/SQ entries), the higher the benefits of applying our technique). In figure 10 we can see that programs *LU-nc*, *ocean-nc* and *radix* show improvements in execution time(by 63.36%, 19.16% and 20.40% respectively) because of less overall stalls as shown in figure 7.2. In particular, *LU-nc* shows an impressive improvement due to 19.70% less stalls in RoB and 77.48% less in the SB/SQ as shown in figure 7.2. All programs except *Barnes, FMM* and *FFT* show an improvement. *FFT* suffers from high processor stalls compared to the baseline – 6% more SB/SQ stalls due to many *xDRF fences*, on average after

every 2.229 *DRF* instructions as can be seen in figure 15. This variation in regions does not allow us to fully utilize the potential of DRF as we have to wait at each *xDRF fence.*) *Barnes* the prefetcher does not perform well, Useful prefetches decreases by 8.84%. For *FMM*, the performance does not improve because of a significant increase in resource stall(5.96%).
Overall, DCSB achieves 10.24% speedup compared to the baseline Store Buffer with 56 entries that implements TSO. Further increases in performance can be obtained with more complex compiler support, such as the technique proposed by Jimborean et al [3] (subsection 7.6). We provide a more in-depth analysis of the performance variation with respect to the SB size using our *DC(Dual Consistency)* technique in subsection 7.4.
A more accurate branch predictor, a more effective prefetcher, or RoB can further improve performance, but we leave this analysis for future research.

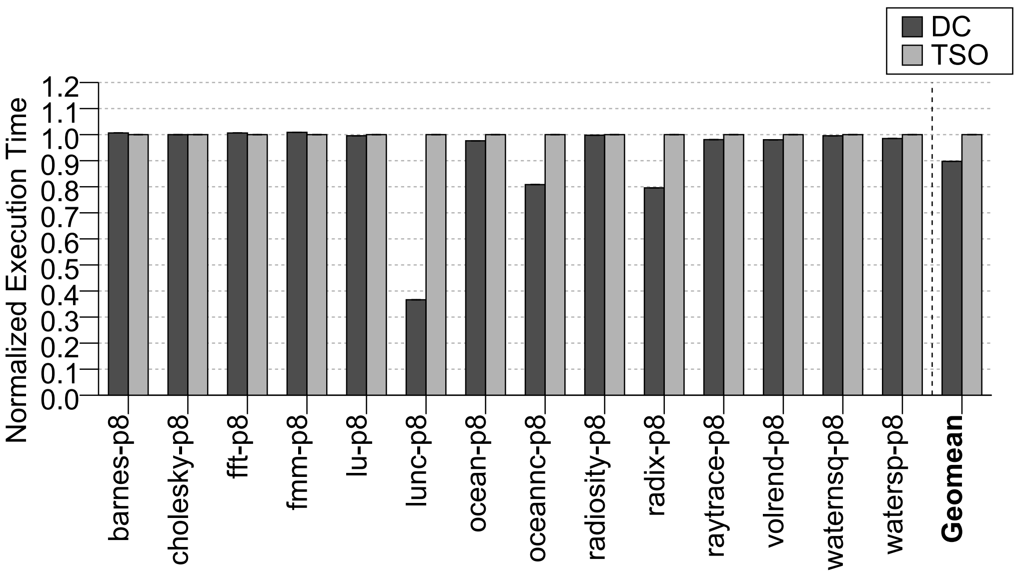


Figure 10: *Normalized execution time with respect to an SB with 56 entries that implements TSO.*

7.2 Processor Stalls

Figure 11 compares three versions:

(1) The baseline: a 56-entry SB with TSO. (2) An oracle version (DRF_All) that sets the *Mode bit* to true permanently. That means, all stores belong to a DRF region and can be performed fully out-of-order. (3) DCSB: our proposal. We provide a breakdown of the stalls in the RoB, LQ, and SQ/SB. SB/SQ stalls are significantly reduced in both DCSB and oracle versions for programs such as *LU-nc*, *ocean-nc* and *radix*, while in other cases the bottleneck moves to either RoB or to LQ. For example, in *ocean-nc* and *LU-nc*, despite reducing the SB/SQ stalls, we observe higher LQ stalls stemming from speculative loads due to miss-branch prediction[11]. In contrast, in *radix* and *water-nsq*, RoB becomes the bottleneck (due to a higher number of instruction squashes, 5.4% more squashes for *radix* and 0.0012% less squashes in *water-nsq*). Other applications, such as *barnes*, *radiosity*, *raytrace* and *volrend* show less RoB stalls (due to fewer instructions being squashed, 2.08%, 1.1%, 1.79% and 2.38% respectively) along with less SB/SQ stalls. *FMM, cholesky* and *water-nsq* perform similarly in terms of stalls in all three versions. *FFT* is the only program that suffers more stalls then TSO, due to the high number of fences, which translates to a slight performance loss, as seen in figure 10. Overall DCSB achieves 8.8% less processor stalls compared to TSO and is almost on par with the oracle, with 8.9% less stalls in *DCSB* and 8.95% in *DRF-All.*

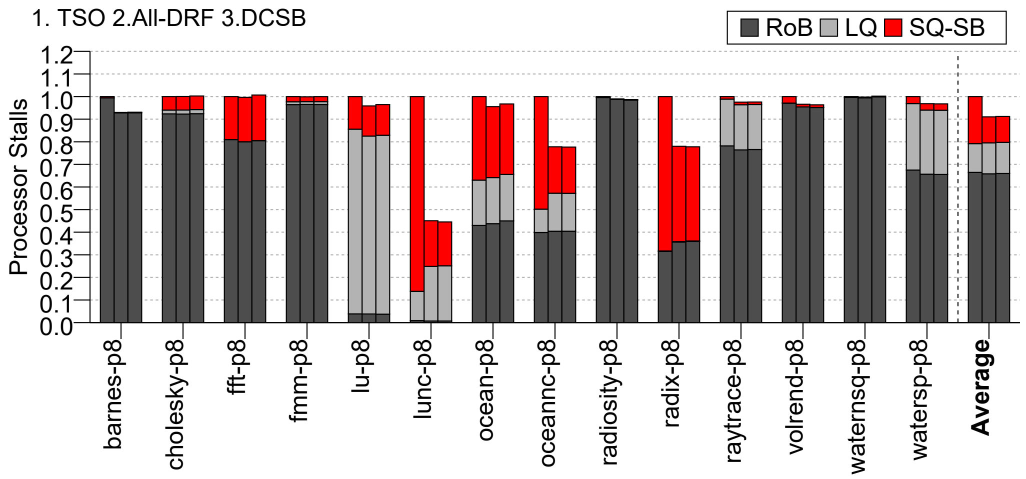


Figure 11: *Processor stalls.*

7.3 Loads forwarded from stores

Keeping the stores in the Store Buffer even after completion increases the number of loads-forwarded-from-store, making the Store Buffer act as a cache. Figure 12 reveals significant improvements in the number of loads-forwarded-from-store in DCSB compared to the baseline. *DCSB* provides outstanding increments in the number of loads-forwarded-from-store, namely 14.83% loads-forwarded-from-store compared to 5.71% in the baseline. The highest percentage is observed in *Barnes* from 11.03% to 44.72% while lowest in *FFT* from 0.0143% to 0.0144%. This finding can be employed in reducing the energy consumption, as showed in the work done by Alves et al [12] where their solution avoids the parallel search in both SB and the cache. We leave this evaluation for future work.

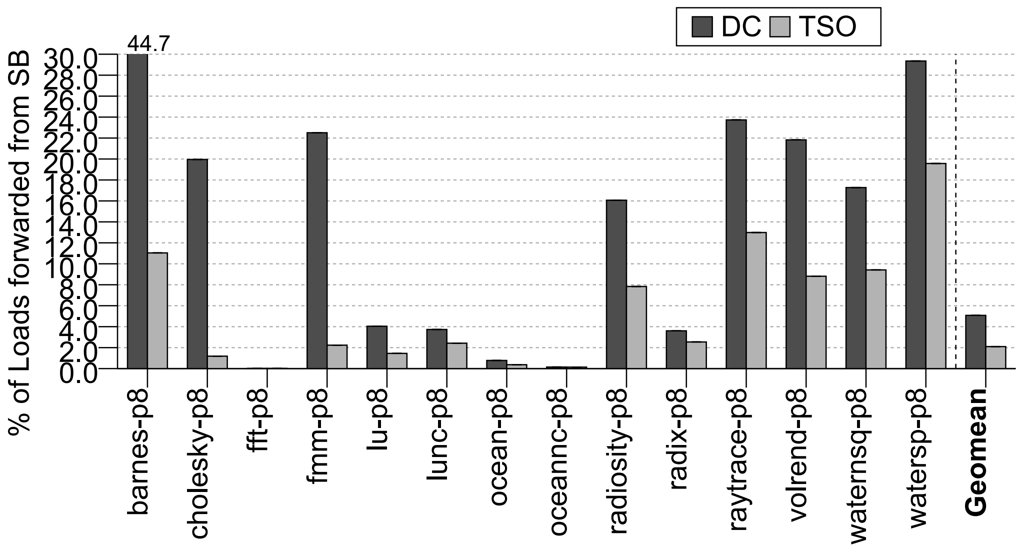


Figure 12: *Percentage of Loads Forwarded from stores.*

7.4 Sensitivity Analysis

Larger SB/SQ buffers do not automatically guarantee performance improvements as data forwarding takes more cycles. If the trend towards higher processor clock, wider pipelines, more execution units and large programs

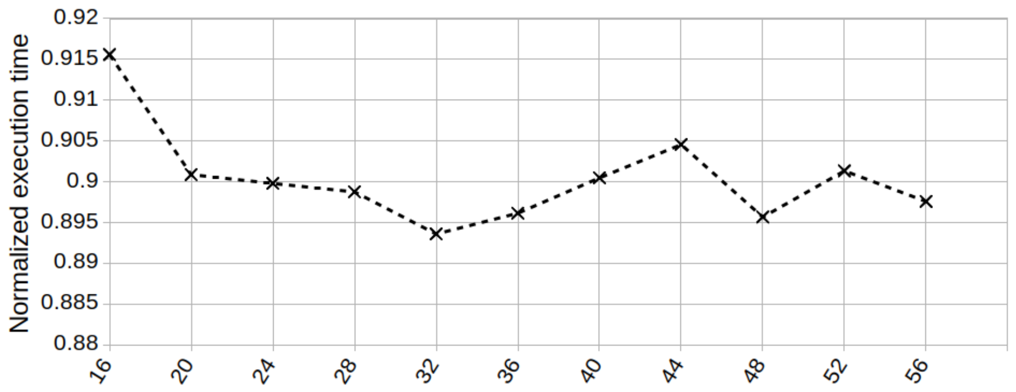


Figure 13: *Normalized execution time for different Store Buffer sizes.*

continues, the latency of search in SB/SQ will become critical for performance. Along with increasing the latency in forwarding data, larger SB/SQ also increases the energy expenditure.

In contrast, a small SB brings numerous benefits such as low power consumption and less hardware overhead, but it affects the overall performance as it increases the SB/SQ stalls. All modern-day processor is a result of a trade-off between energy expenditure and speed. Our analysis shows that we can achieve low energy expenditure without degrading performance, by boosting performance even with a small SB. Figure 13 shows the normalized execution time (average of all applications from the SPLASH-3 benchmark suite) when varying the SB size. The baseline is a Store Buffer of 56 entries implementing TSO (as before). Figure 14 shows the performance of each benchmark from SPLASH-3. Even with a Store Buffer as small as 16 entries, DCSB leads to performance improvements of 8.4%. Overall, a Store Buffer with 32 entries gives the highest performance of 10.64%.

DCSB becomes particularly attractive in the context of a growing demand for low power and high performance, especially since it requires minimal modifications of the mainstream SB implementations.

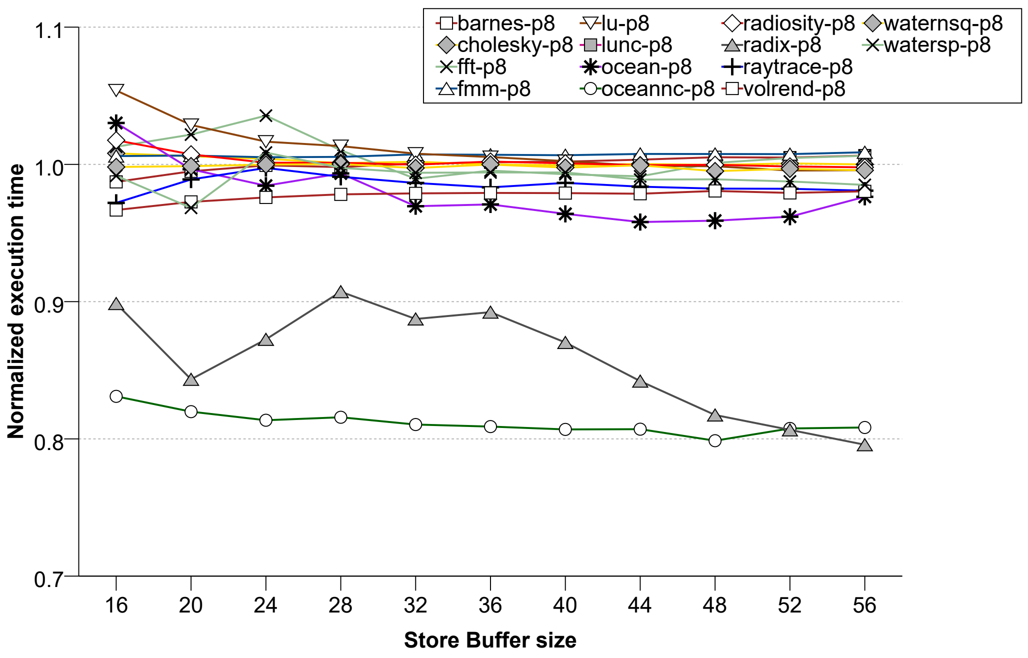


Figure 14: *Normalized execution time for SPLASH-3.*

7.5 xDRF fences

DCSB relies on *xDRF fence* to ensure correctness. Figure 15 shows the percentage of *xDRF fences* with respect to the total number of memory operations. Overall, the compiler inserts 9.39% instructions that are fenced to provide the TSO guarantees of sequential semantic and consistency. *Radix* exhibits the highest percentage of *xDRF fences*, 56.34%. Being logical fences, the *xDRF fence* does not require any special resource or hardware to execute. Although, for performance reasons, a lower number of *xDRF fences*, i.e. larger *DRF regions*, provides higher potential for store reordering. Each fence imposes waiting until all ongoing memory operations are finished, to avoid inter-region reordering, which can also lead to performance degradations, as in the case of *FFT* in figure 10.

7.6 More complex compiler support

xDRF which stands for *Extended Data Race Free* is a compiler technique developed by Jimborean et al [13]. *xDRF* are sets of DRF regions

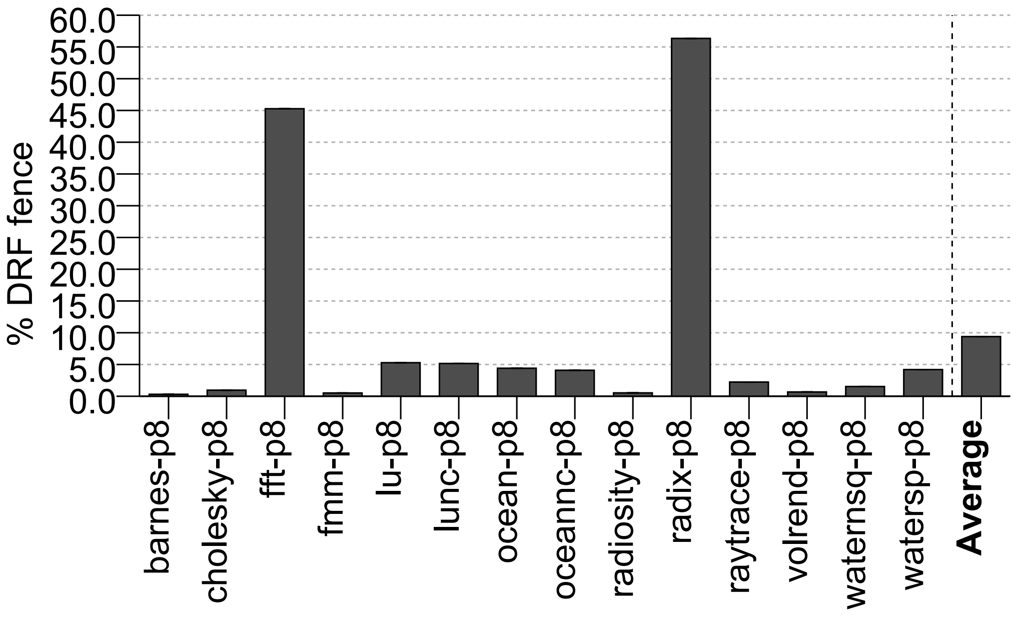


Figure 15: *% of xDRF fences in various benchmarks*

extend across synchronization points, function calls, loops, etc while providing the same guarantee as a synchronization-free region. DCSB applied on larger *DRF regions* leads to improvements of about 1% compared to synchronization-free regions (DRF as described in this paper) and improvement of 11.72% compared to the baseline SB (TSO with 56 entries). More complex compiler support can increase performance with minimal or zero hardware costs.

8 Related work

A plethora of techniques have been proposed for efficient hardware designs that provide Sequential Consistency guarantees [14] [15] [16] [17] [18]. They rely on speculation to enable a more efficient TSO model and require support to recover upon SC violations. Speculation occurs before the stores leave the RoB. The same technique is also used in commercial processors like x86 [19]. This optimization alone is insufficient to obtain high performance, thus more aggressive speculation techniques are proposed [15] [18]. The more aggressive the speculation, the more costly becomes to check and recover from SC violations. The required mechanism is quite complex and increases the hardware overhead (e.g. through extra registers to track the speculation). In consequence, none of these techniques is implemented in any real processor because of the added complexity.

We detail in what follows a selection of the most relevant techniques.

Singh *et al.* [20] proposes a software-hardware co-design in which the compiler provides information about which accesses can be reordered. Their design implements two Store Buffers (out-of-order for safe stores and in-order for unsafe ones, respectively) and store operations are allocated in one or the other depending on their type. This design with two Store Buffers increases the complexity of the solution and adds a very high hardware overhead (e.g. for the logic unit that selects the Store Buffer to scoop for the forwarded data). Also, if a program consists of numerous shared stores or vice-versa (unshared stores), this design can lead to under-utilization of the hardware along with generating numerous stalls in the corresponding Store Buffer. Since in our design the processor uses the same Store Buffer, the hardware is always utilized effectively and more efficiently. Our design also has considerably less overhead in terms of added hardware, yielding DCSB ready to be easily integrated into current processors' design, a key feature of our solution.

Moreover, the compiler support employed by Singh *et al.* [20] forces all accesses to the same memory address to be classified as having the same type and to be assigned the same Store Buffer to solve the notorious store-store reordering and store-to-load forwarding problems. Due to this limitation, this approach only enables reordering of 75% of the stores, on average. For example, they reorder 50% of the stores in Barnes, while DCSB enables the reordering of more than 90% of the stores, thanks to exploiting the DRF semantics of the code.

Ros and Kaxiras [21] propose coalescing stores in the SB and avoid breaking the store order by performing stores in atomic groups. In contrast to our approach where DRF stores can perform completely out of order, stores in an atomic group perform following a globally defined order. On the other hand, coalescing stores can be applied to DCSB (either for DRF stores or NDRF stores). This can further improve our performance as coalescing may reduce the occupancy of the SB. The counterpart of this approach is that it requires a separate store queue and Store Buffer to fully get the benefits of coalescing.

Another interesting research proposal was done by Alves *et al.* [12], for using the Store Buffer as a cache. This significantly increases the number of loads forwarded from stores. In DCSB we archive the same without using any extra hardware for this specific purpose. Alves *et al.* [12] predict in advance the hits, thus reducing the number of L1 accesses resulting in less energy consumption. In our design, we increase the number of loads forwarded from stores significantly by storing the stores in the SB as long as possible (until the Store Buffer is full or until an atomic operation is encountered. One simple extension to predict the hit can be used to reduce the power consumption along with an increase in speedup.

9 Conclusion

Relaxed Stores can go out of order thus reducing the stall drastically. Before issuing the store we start from the head and move to tail while trying to perform store. If we are unable to issue to store due to some reason then we check whether it is NDRF or DRF if its NDRF we try to issue the same store again and again until its issued. While in the case of DRF we try to issue the next store to the head and so on. If our store buffer has xDRF fences we stop issuing store and thus we wait for all the stores to performed to start issuing them again. This helps us to maintain consistency as if there are DRF stores after the xDRF fence and somehow they are performed earlier. This will break the consistency as there are stores that should be performed earlier because no store should be reordered outside the fence. In the case of miss we unlock, remove all the NDRF stores except the one at the head. All the unlocked stores will be issued again. While once the DRF is issued they are marked performed and are ready to dequeue from the store buffer as even if they give a miss they can be reordered so they will try again and will perform in the end. While at every xDRF fence we wait until all the stores including the NDRF and DRF are done performing the cache thus this keeps these sequential properties. This concludes our overall work and in the result, we can see the improvement. It is a win-win situation as the overall hardware overhead will be just one bit per entry in the store buffer to keep the DRF information which will be in few bits.

Overall improvements are, 10.24% more performance(11.72% using xDRF compiler), 8.95% fewer stalls, loads forwarded from stores increased from 5.71% to 14.83% and the most important, achieving a speedup of 8.4% with very small store buffer of size 16 entries.

References

[1] J. Bornholt, “Memory Consistency Models: A Tutorial,” https://www.cs.utexas.edu/ bornholt/post/memory-models.html, Feb. 2016.

[2] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors,” *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[3] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros, “Automatic detection of extended data-race-free regions,” in *15th Int’l Symp. on Code Generation and Optimization (CGO)*, Feb. 2017, pp. 14–26.

[4] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations,” in *Conf. on Supercomputing (SC)*, Nov. 2011, pp. 52:1–52:12.

[5] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, “Splash-3: A properly synchronized benchmark suite for contemporary research,” in *Int’l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[6] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *22nd Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *2005 Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[8] S. V. Adve and M. D. Hill, “Weak ordering – a new definition,” in *17th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[9] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,”

ACM SIGARCH Computer Architecture News, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[10] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[11] A. Ros and S. Kaxiras, "The superfluous load queue," in *51st IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2018.

[12] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras, "Filter caching for free: The untapped potential of the store buffer," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 436–448.

[13] A. Jimborean, P. Ekemark, J. Waern, S. Kaxiras, and A. Ros, "Automatic detection of large extended data-race-free regions with conflict isolation," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 3, pp. 527–541, Mar. 2018.

[14] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong, "BulkCompiler," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42.* ACM Press, 2009.

[15] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 278–289.

[16] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[17] M. D. Hill, "Multiprocessors should support simple memory-consistency models," *IEEE Computer*, vol. 31, no. 8, pp. 28–34, Aug. 1998.

[18] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 266–277.

[19] "Intel Corporation," http://www.intel.com, [Online; accessed Jan-2016].

[20] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, “End-to-end sequential consistency,” in *39th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[21] A. Ros and S. Kaxiras, “Non-speculative store coalescing in total store order,” in *45th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 221–234.

A Configuration File

```
      Ruby Configuration
------------------
protocol: MESI_CMP_directory_inclusive
compiled_at: 12:57:49, Nov 20 2019
RUBY_DEBUG: true
hostname: ng11
SIMULATION_COMMENT: s
SIMULATION_BENCHMARK: fft-p8
g_RANDOM_SEED: 1
g_DEADLOCK_THRESHOLD: 500000
RANDOMIZATION: false
g_SYNTHETIC_DRIVER: false
g_DETERMINISTIC_DRIVER: false
g_FILTERING_ENABLED: false
g_DISTRIBUTED_PERSISTENT_ENABLED: true
g_DYNAMIC_TIMEOUT_ENABLED: true
g_RETRY_THRESHOLD: 1
g_FIXED_TIMEOUT_LATENCY: 300
g_trace_warmup_length: 1000000
g_bash_bandwidth_adaptive_threshold: 0.75
g_tester_length: 0
g_synthetic_locks: 2048
g_deterministic_addrs: 1
g_SpecifiedGenerator: DetermGETXGenerator
g_NUM_COMPLETIONS_BEFORE_PASS: 0
g_NUM_SMT_THREADS: 1
g_think_time: 5
g_hold_time: 5
g_wait_time: 5
PROTOCOL_DEBUG_TRACE: true
PROTOCOL_DEBUG_TRACE_MESSAGES: false
TRACE_TYPE: pthreads
DEBUG_FILTER_STRING: none
DEBUG_VERBOSITY_STRING: none
DEBUG_START_TIME: 0
DEBUG_OUTPUT_FILENAME: none
```

```
TRANSACTION_TRACE_ENABLED: false
USER_MODE_DATA_ONLY: false
PROFILE_HOT_LINES: false
PROFILE_ALL_INSTRUCTIONS: false
PRINT_INSTRUCTION_TRACE: false
INSTRUCTIONS_SIMULATED: 0
VIRTUAL_L1CACHES: false
PROFILE_SYNONYMS: false
CLEAR_PROFILE_SYNONYMS: true
g_DEBUG_CYCLE: 0
PERFECT_MEMORY_SYSTEM: false
PERFECT_MEMORY_SYSTEM_LATENCY: 0
DATA_BLOCK: false
REMOVE_SINGLE_CYCLE_DCACHE_FAST_PATH: true
PROCESSOR_MODEL: OoO
PROCESSOR_FETCH_WIDTH: 5
PROCESSOR_DECODER: 1_1_1_1_4|6
PROCESSOR_COMMIT_WIDTH: 10
PROCESSOR_ALLOCATION_QUEUE_SIZE: 97
PROCESSOR_ROB_SIZE: 224
PROCESSOR_LOAD_QUEUE_SIZE: 72
PROCESSOR_STORE_QUEUE_SIZE: 0
PROCESSOR_STORE_BUFFER_SIZE: 56
PROCESSOR_EXECUTION_PORTS: 8
PROCESSOR_ALU_LATENCY: 1
PROCESSOR_ALU_RECIPROCAL_THROUGHPUT: 0.25
PROCESSOR_AGU_LATENCY: 1
PROCESSOR_AGU_RECIPROCAL_THROUGHPUT: 0.25
PROCESSOR_FP_ADDSUB_LATENCY: 4
PROCESSOR_FP_ADDSUB_RECIPROCAL_THROUGHPUT: 0.5
PROCESSOR_FP_MUL_LATENCY: 4
PROCESSOR_FP_MUL_RECIPROCAL_THROUGHPUT: 0.5
PROCESSOR_FP_DIV_LATENCY: 14
PROCESSOR_FP_DIV_RECIPROCAL_THROUGHPUT: 8
PROCESSOR_FP_SQRT_LATENCY: 16
PROCESSOR_FP_SQRT_RECIPROCAL_THROUGHPUT: 8
PROCESSOR_BRANCH_LATENCY: 1
PROCESSOR_BRANCH_RECIPROCAL_THROUGHPUT: 0.5
```

```
PROCESSOR_STORE_BUFFER_AS_CACHE: true
PROCESSOR_CONSISTENCY: TSO
PROCESSOR_IMPLEMENTS_UOP_FUSION: true
PROCESSOR_IMPLEMENTS_MACRO_FUSION: true
PROCESSOR_BRANCH_PREDICTOR: LTAGE
PROCESSOR_MEMORY_DISAMBIGUATION: SpeculateAndSnoop
PROCESSOR_MEMORY_DEPENDENCE_PREDICTOR: TaglessCHT
PROCESSOR_MEM_DEP_PRED_BITS_PC: 12
PROCESSOR_MEM_DEP_PRED_BITS_COUNTER: 2
PROCESSOR_MEM_DEP_PRED_MAXCYCLES: 100000
PROCESSOR_MEM_DEP_PRED_BITS_LFST: 9
PROCESSOR_ONLY_REEXEC_DSPEC_ON_ALIAS: true
PROCESSOR_STORE_ATOMICITY_SPECULATIVE: No
PROCESSOR_EAGER_SQUASH: true
PROCESSOR_EAGER_REEXEC: true
PROCESSOR_REEXEC_DO_SQUASH: false
PROCESSOR_STORE_BUFFER_TYPE: Hybrid
PROCESSOR_STORE_BUFFER_ISSUE_POLICY: ASAP_head
PROCESSOR_STORE_BUFFER_ISSUE_TIMEOUT: 2000
PROCESSOR_STORE_BUFFER_ISSUE_OCCUPANCY: 16
PROCESSOR_STORE_BUFFER_KEEP_L0_ENTRIES: false
PROCESSOR_STORE_BUFFER_IS_COLLAPSIBLE: false
PROCESSOR_STORE_PREFETCH: OnCommit
PROCESSOR_STORE_PREFETCH_FILTER_CSPEC: false
PROCESSOR_UNIFIED_STORE_QUEUE_AND_BUFFER: true
PROCESSOR_COMMIT_TYPE: IO
PROCESSOR_OOO_COMMIT_RESPECT_UNKNONW_STORE_ADDRESS: true
PROCESSOR_ALLOW_REEXEC_IN_NONREEXEC_PROTOCOLS: false
PROCESSOR_LOCKDOWN_TABLE_ENTRIES: 16
CSB_CACHE_NUM_SETS_BITS: 6
CSB_CACHE_ASSOC: 8
CSB_NUM_CACHES: 2
g_NUM_PROCESSORS: 8
g_NUM_L2_BANKS: 8
g_NUM_MEMORIES: 8
g_NUM_L1S_BANKS: 8
g_PROCS_PER_CHIP: 8
g_NODES_PER_FPGA: 1
```

```
L0_CACHE_ENABLE: true
L0_CACHE_ASSOC: 8
L0_CACHE_NUM_SETS_BITS: 6
L0_CACHE_ACCESS_LATENCY: 4
L0_CACHE_PREFETCHER: Stride
L0_CACHE_PREFETCH_THROTTLING: 3
L0_CACHE_PREFETCH_QUEUE_SIZE: 8
L1_CACHE_ASSOC: 8
L1_CACHE_NUM_SETS_BITS: 9
L2_CACHE_ASSOC: 16
L2_CACHE_NUM_SETS_BITS: 10
L3_CACHE_ASSOC: 4
L3_CACHE_NUM_SETS_BITS: 16
PF_CACHE_ASSOC: 4
PF_CACHE_NUM_SETS_BITS: 16
EPF_CACHE_ASSOC: 16
EPF_CACHE_NUM_SETS_BITS: 16
DIR_CACHE_ASSOC: 16
DIR_CACHE_NUM_SETS_BITS: 9
DIR_CACHE_STORE_NULLS: false
CBDIR_CACHE_ASSOC: 16
CBDIR_CACHE_NUM_SETS_BITS: 2
L1_TLB_ASSOC: 32
L1_TLB_NUM_SETS_BITS: 8
L1_SDS_ASSOC: 4
L1_SDS_NUM_SETS_BITS: 8
L1_SDS_STORE_NULLS: false
L1_SHARED_CACHE_ASSOC: 4
L1_SHARED_CACHE_NUM_SETS_BITS: 5
L2_INCLUSIVE: true
g_MEMORY_SIZE_BYTES: 4294967296
g_DATA_BLOCK_BYTES: 64
g_PAGE_SIZE_BYTES: 4096
g_REPLACEMENT_POLICY: LRU
L1D_INDEXING_POLICY: LSB
L1I_INDEXING_POLICY: LSB
L2_INDEXING_POLICY: LSB
g_L2BANKS_MAPPING_GRANULARITY: 0
```

```
g_MEMORY_MAPPING_GRANULARITY: 0
g_SHARING_CODE: CoarseVector
g_SHARING_CODE_VALUE: 1
g_USE_SPECIAL_NODE_LAYOUT: false
TLB_CACHE_INCLUSION: true
DEACTIVATED_PAGES: None
TEMPORALITY_TLB_SUPPORT: None
DECAY_TIMEOUT: 10000
FAST_TLB_MISS_RESOLUTION: false
FENCING_STRATEGY: 28
g_THREAD_MIGRATION: true
g_THREAD_MIGRATION_TIMEOUT: 100
g_LOCK_TYPE: TATAS
g_BARRIER_TYPE: SRNoLock
g_COND_TYPE: Spin
g_EXPHW_TYPE: Spin
g_CALLBACK_MECHANISM: None
g_EXP_BACKOFF_LIMIT: 0
g_SLE_POLICY: All
DIRECTORY_CACHE_LATENCY: 2
NULL_LATENCY: 1
ISSUE_LATENCY: 2
CACHE_RESPONSE_LATENCY: 12
L2_RESPONSE_LATENCY: 36
L2_TAG_LATENCY: 6
L1_RESPONSE_LATENCY: 12
MEMORY_RESPONSE_LATENCY_MINUS_2: 98
DIRECTORY_LATENCY: 2
FPGA_LATENCY: 20
NETWORK_LINK_LATENCY: 1
COPY_HEAD_LATENCY: 4
ON_CHIP_LINK_LATENCY: 1
RECYCLE_LATENCY: 1
L2_RECYCLE_LATENCY: 5
TIMER_LATENCY: 10000
TBE_RESPONSE_LATENCY: 1
L1_SHARED_REQUEST_LATENCY: 4
L1_SHARED_RESPONSE_LATENCY: 4
```

```
L2_DIRECTORY_REQUEST_LATENCY: 2
L2_DIRECTORY_RESPONSE_LATENCY: 2
TLB_MISS_LATENCY: 1000
POST_TLB_HIT_LATENCY: 0
SYNONYMS_CHECKING_LATENCY: 0
REVERSE_TRANSLATION_LATENCY: 0
PROFILE_EXCEPTIONS: false
PROFILE_XACT: true
PROFILE_NONXACT: false
XACT_DEBUG: true
XACT_DEBUG_LEVEL: 1
XACT_MEMORY: false
XACT_ENABLE_TOURMALINE: false
XACT_NUM_CURRENT: 0
XACT_LAST_UPDATE: 0
XACT_ISOLATION_CHECK: true
PERFECT_FILTER: true
READ_WRITE_FILTER: Perfect_
PERFECT_VIRTUAL_FILTER: true
VIRTUAL_READ_WRITE_FILTER: Perfect_
PERFECT_SUMMARY_FILTER: true
SUMMARY_READ_WRITE_FILTER: Perfect_
XACT_EAGER_CD: true
XACT_LAZY_VM: false
XACT_CONFLICT_RES: BASE
XACT_VISUALIZER: false
XACT_COMMIT_TOKEN_LATENCY: 0
XACT_NO_BACKOFF: false
XACT_LOG_BUFFER_SIZE: 0
XACT_STORE_PREDICTOR_HISTORY: 256
XACT_STORE_PREDICTOR_ENTRIES: 256
XACT_STORE_PREDICTOR_THRESHOLD: 4
XACT_FIRST_ACCESS_COST: 0
XACT_FIRST_PAGE_ACCESS_COST: 0
ENABLE_MAGIC_WAITING: false
ENABLE_WATCHPOINT: false
XACT_ENABLE_VIRTUALIZATION_LOGTM_SE: false
ATMTP_ENABLED: false
```

```
ATMTP_ABORT_ON_NON_XACT_INST: false
ATMTP_ALLOW_SAVE_RESTORE_IN_XACT: false
ATMTP_XACT_MAX_STORES: 32
ATMTP_DEBUG_LEVEL: 0
L1_REQUEST_LATENCY: 12
L2_REQUEST_LATENCY: 36
CACHE_READ_PORTS: 2
CACHE_WRITE_PORTS: 1
L1CACHE_HIT_LATENCY: 12
L1CACHE_HIT2_LATENCY: 12
L1CACHE_MISS_LATENCY: 12
L1CACHE_MISS2_LATENCY: 12
L1_SHARED_CACHE_HIT_LATENCY: 12
L1_SHARED_CACHE_HIT2_LATENCY: 12
L1_SHARED_CACHE_MISS_LATENCY: 12
L1_SHARED_CACHE_MISS2_LATENCY: 12
L2CACHE_HIT_LATENCY: 36
L2CACHE_HIT2_LATENCY: 36
L2CACHE_MISS_LATENCY: 36
L2CACHE_MISS2_LATENCY: 36
L1CACHE_TRANSITIONS_PER_RUBY_CYCLE: 10000
L2CACHE_TRANSITIONS_PER_RUBY_CYCLE: 10000
DIRECTORY_TRANSITIONS_PER_RUBY_CYCLE: 10000
FPGA_TRANSITIONS_PER_RUBY_CYCLE: 10000
g_SEQUENCER_OUTSTANDING_REQUESTS: 64
NUMBER_OF_L1_TBES: 128
NUMBER_OF_L2_TBES: 128
NUMBER_OF_WRITTEN_BITS_ENTRIES: 16
BLOOM_FILTER_TYPE: Perfect_6
FINITE_BUFFERING: false
FINITE_BUFFER_SIZE: 3
PROCESSOR_BUFFER_SIZE: 32
PROTOCOL_BUFFER_SIZE: 32
g_NETWORK_MODEL: Garnet-fixed
g_NETWORK_TOPOLOGY: FILE_SPECIFIED
g_SICOSYS_CONFIG: M84-WH-H1-VC-2C
g_SICOSYS_UNICAST_NETWORK: true
g_SICOSYS_OFF_CHIP_LATENCY: 40
```

```
g_SICOSYS_RUBY_NETWORK_MULTIPLIER: 1
g_CACHE_DESIGN: MESH_2D_3DMEM_1cycle
g_NETWORK_FILES_DIRECTORY: network/simple/Network_Files
g_endpoint_bandwidth: 1000
g_adaptive_routing: true
NUMBER_OF_VIRTUAL_NETWORKS: 5
FAN_OUT_DEGREE: 4
g_PRINT_TOPOLOGY: false
XACT_LENGTH: 2000
XACT_SIZE: 1000
ABORT_RETRY_TIME: 400
g_NETWORK_TESTING: false
g_FLIT_SIZE: 16
g_NUM_PIPE_STAGES: 4
g_VCS_PER_CLASS: 4
g_BUFFER_SIZE: 8
NETWORK_LINK_LATENCY_MULTIPLIER_IN: 1
NETWORK_LINK_LATENCY_MULTIPLIER_OUT: 1
NETWORK_LINK_LATENCY_MULTIPLIER_INTERNAL: 1
MEM_BUS_CYCLE_MULTIPLIER: 10
BANKS_PER_RANK: 8
RANKS_PER_DIMM: 2
DIMMS_PER_CHANNEL: 2
BANK_BIT_0: 8
RANK_BIT_0: 11
DIMM_BIT_0: 12
BANK_QUEUE_SIZE: 12
BANK_BUSY_TIME: 11
RANK_RANK_DELAY: 1
READ_WRITE_DELAY: 2
BASIC_BUS_BUSY_TIME: 2
MEM_CTL_LATENCY: 12
REFRESH_PERIOD: 1560
TFAW: 0
MEM_RANDOM_ARBITRATE: 0
MEM_FIXED_DELAY: 0
NUMBER_OF_TAGS: 32
NUMBER_OF_FULL_TAGS: 32
```

```
EXTENDED_TAG_TABLE_ENTRIES: 64
g_USE_TIMEOUT: 50
g_CHECKRACE_TIMEOUT: 50
MANDATORY_DEQUEUES_PER_CYCLE: 999999999
L1_DEQUEUES_PER_CYCLE: 999999999
L2_DEQUEUES_PER_CYCLE: 999999999
DIRECTORY_DEQUEUES_PER_CYCLE: 999999999
MANDATORY_DEQUEUE_LATENCY: 0
L1_DEQUEUE_LATENCY: 0
L2_DEQUEUE_LATENCY: 0
DIRECTORY_DEQUEUE_LATENCY: 0
LINKED_DIR_ALWAYS_PREPEND: true
LINKED_DIR_SEND_DATA_FROM_SHARER: false
LINKED_DIR_ALLOW_SELF_REFERENCES: false
LINKED_DIR_TRY_DIRECT_PUTS: false
LINKED_DIR_OPORTUNISTIC_PUTS: true
LINKED_DIR_CONCURRENT_GETS_PUTS: true
SCD_REPLACEMENT_LEVELS: 2
SCD_LATENCY_PER_EXTRA_ACCESS: 2
SCD_REPLACEMENT_POLICY: LRU
WAYCOMB_MAX_COMBINED_WAYS: 0
WAYCOMB_REPLACEMENT_POLICY: CompactMRU
WAYCOMB_BROADCAST_FORCES_SILENT_S_REPLACEMENTS: false
SHARING_CODE_POINTERS_TO_COARSEVECTOR: true
SILENT_S_REPLACEMENTS: true
DIRECTORY_SNAPSHOT_PERIOD: 0
g_NUM_CHIPS: 1
g_NUM_CHIP_BITS: 0
g_MEMORY_SIZE_BITS: 32
g_DATA_BLOCK_BITS: 6
g_PAGE_SIZE_BITS: 12
g_NUM_PROCESSORS_BITS: 3
g_PROCS_PER_CHIP_BITS: 3
g_NUM_L2_BANKS_BITS: 3
g_NUM_L2_BANKS_PER_CHIP_BITS: 3
g_NUM_L2_BANKS_PER_CHIP: 8
g_NUM_L1S_BANKS_BITS: 3
g_NUM_L1S_BANKS_PER_CHIP_BITS: 3
```

```
g_NUM_L1S_BANKS_PER_CHIP: 8
g_NUM_MEMORIES_BITS: 3
g_NUM_MEMORIES_PER_CHIP: 8
g_MEMORY_MODULE_BITS: 23
g_MEMORY_MODULE_BLOCKS: 8388608
g_NUM_FPGAS: 8
g_NUM_FPGAS_PER_CHIP: 8
g_BOARD_BITS_OFFSET: 29
g_NODE_BITS_OFFSET: 29
g_L2_BANK_MAPPING_TAG_INIT_BIT: 6
g_NUMBER_OF_BITS_LEX_ORDER: 9
```