Improving the Performance of Concurrent Operations utilizing the memory hierarchy

By : Satish Dinakar B , Computer Science Automation, IISc .
Kaushik B.V , Computer Science Automation, IISc

Abstract:

The access time of RAM (physical memory) is very slow compared to the execution time of CPU. To effectively utilize the latency of memory has always been a very a very challenging problem for computer architect scientists since the dawn of microprocessor era. In this project we propose to utilize the memory latency time to eliminate cancelling concurrent operations. We propose new instruction and certain changes in architecture so as to eliminate concurrent operation across multi-cores when they are in turn waiting for a memory operation.

1. Introduction

Time per CPU operation is less than one nanosecond, whereas the average memory operation approximately takes around 10’s of nano-seconds. The memory stalls are seldom utilized effectively. In this project we propose to utilized the memory stall cycles by carefully cancelling certain concurrent operations by pairing eliminating/ combining operations.

In the following subsections we introduce two concurrent programming paradigms widely studied in the literature the Combing-Software tree approach & the Lock-free paradigm. Later we combine these 2 paradigms to provide concurrent cancelling instructions.

1.1 Combing tree paradigm

A CombiningTree is a binary/n-ary tree of *nodes*, where each node contains bookkeeping information. We illustrate the method of software combing using the example of shred counter. The counter’s value is stored at the root. Each thread is assigned a leaf, and at most two threads share a leaf, so if there are *p* physical processors, then there are *p/*2 leaves. To increment the counter, a thread starts at its leaf, and works its way up the tree to the root. If two threads reach a node at approximately the same time, then they *combine* their increments by adding them together. One thread, the *active* thread, propagates their combined increments up the tree, while the other, the *passive* thread, waits for the active thread to complete their combined work. A thread may be active at one level and become passive at a higher level. The whole process is repeated back from root to leaf, and it distributes the work back as it traverses back to leaf. In the best case we get a speed up of n/ log n.

The software combing tree has its own setoff disadvantages; it performs very poorly when there is comparatively less contention in the sense it performs as poorly giving a bottleneck of O(n log n).

1.2 Lock-free para digm

Lock-free paradigm tries to implement concurrent data structures using special instructions such as compareAndSwap(). The advantage of this approach is that even though there might be one or two process starving, there is always a notion of the whole program moving in a progressive manner.

The disadvantage with this approach is, that it does not perform well when there is a high contention, which could be marginally increased with the help of back-off and elimination techniques, but cannot in anyway match the performance improvement in the combining paradigm for the same problems under high contention,

2. Proposed Solution

We propose to combine the lock -free and combing tree paradigms to provide a new instruction SCAS() which uses the memory stall cycle to cancel concurrent memory operations. To add the new instruction we propose the following change in the architecture changes:

We add three special registers per core namely DATA-REGISTER, PRO CESS-ID-REGISTER, and STATE-REGISTER. Schematically, the Operating System while scheduling a particular process will store the process-id in the PROC ESS-ID-REGISTER. The STATE-REGISTER and DATA-REGISTERS are general purpose registers, the STATE-REGISTER register is primarily used by the programmer to store the state of the concurrent operation that is being executed in the processor, for example, a thread which is pushing an element in a concurrent stack will try to reflect in the STATE-R EGISTER that it is in a pushing state, and similarly a popping thread would store a value in STATE-REGISTER that would reflect that it is in the popping state. The DATA-REGISTER is used to store/ retrieve the data that could potentially be eliminated, for example the thread which is trying to push an element in the concurrent stack will place the element to be push ed in the DATA-REGISTER waiting for a cancelling pop operation. The figure below illustrates the above said statements:

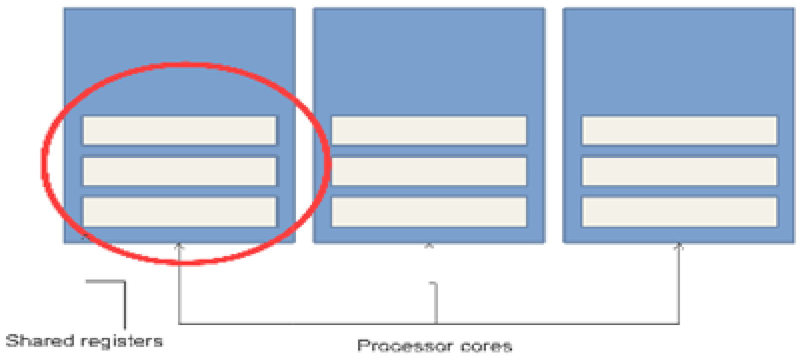

Figure 1: Multiple processors with shared registers

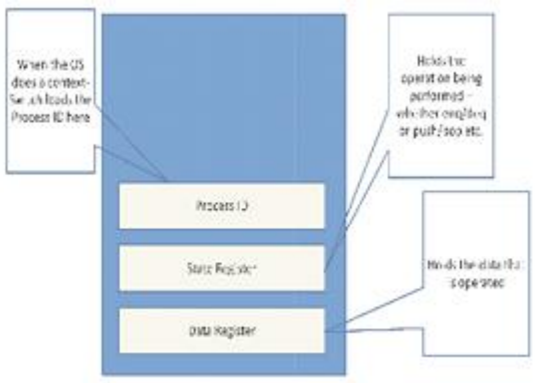
When the OS does a context-Switch loads the Process ID here
Holds the operation being performed - whether enq/deq or push/pop etc.
Process ID
State Register
Data Register
Holds the data that's operated

Figure 2: Three Registers added that are added

2.1 SCAS()

We know propose a new instruction SCAS() which makes use of the three new proposed register, and tries to cancel the concurrent ope rations during a memory stall. The proposed instruction is split into 3 parts, 2 parts of it which is customizable by the programmer, namely preConditionStage, NormalCASCall & cancellationCondition. The preConditionStage and the cancellationContionStage are the entities that are programmable by programmer, in the preConditionStage the programmer basically sets up the shared registers indicating that it is read y to get eliminated, the cancellation ConditionStage is the part where each processor tries to find out a complementary/ combining pair with all other c ores. The structure of SCAS could be visually described in the figure below:

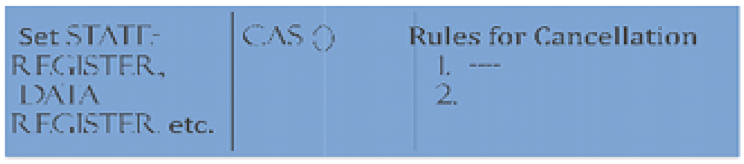

Figure 3: Structure of SCAS

In the cancellationConditionStage for cancellation to occur the programmer can use CAS2 (), Cas2 () compares two registers STATE- REGISTER and PROCESS-ID-REGISTER If both have the specified values, then they swap the STATE-REGISTER and DATA-REGISTER appropriately. If cancellation notion succeeds then the Normal CAS () would be aborted, the status register of the core would reflect that the success of Cancellation.

At the end of the SCAS instruction the programmer has to check the possible outcomes of the SCAS:

a> The SCAS returns false, implying both CAS as well as an attempted elimination failed.

b> The CAS returns true, implying that the normal CAS has succeeded.

c> The cancellation/combining has taken place; in this case the programmer may have to fetch the operands from DATA/STATE REGISTER.

The SCAS instruction could be abstracted as a java interface as follows:

```
public interface SCASintf {
  public preCondition();
  public cancellationCondition( int coreNumber);
}
```

Each thread that has to execute the SCAS has to define the SCASintf before it calls the SCAS(). The algorithm for the SCAS() could be outlined as follows:

```
// Instruction SCAS

public instruction SCAS() {
      core[this].preCondtion();
      // The following would be done during
     // memory stall while actual CAS is being
     //executed
      for (int i=0; i<numOfProcessors;i++ )
           if (core[this].cancellationCondition(i)){
                             set status registers
                            flush the CAS operation
                            break;
           }
 }
```

We can use the above semantics of SCAS() to execute many concurrent data structures such as stacks, queues, lists, shared counters etc.. We illustrate one such example for stack with very minor modifications to the lock-free stack as shown in "The Art of Multiprocessor programming" by Herlihy et.al..

```
//Example Lock-free implementation using SCAS

public class LockFreeStack implements SCASintf {
      private AtomicReference top =
       new AtomicReference(null);
        public boolean tryPush(Node node){
      Node oldTop = top.get();
      node.next = oldTop;
      ret=top.SCAS(oldTop, node);

      if (status.getCancellationFlag())

                return true;
        if ( ret )// CAS succeeded
                return true;
        return  false;
  }
public void preCondition() {
    set STATE-REGISTER as push;
           set DATA-REGISTER the item to be pushed;
}
  public boolean cancellationCondition( int
     coreNumber){ expectedSTATE-REG= POP;
     if CAS2( coreNumber)
                return true;
     else
         return false;
 }
```

3. Simulation Models for Proposed Solution

3.1 Manual Event Driven Simulation of instructions

First we tried to simulate the above set up using a manual event driven simulation of instructions. For the above proposal we had to simulate the instruction execution, cache system, memory system. It was difficult for us to come up with such a model which could be implemented in the allotted time for project.

3.2 Modifying M5 simulator

M5 is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor micro architecture, i.e. the simulator simulates the whole system including CPU, Cache memory, Physical Memory (RAM). We had to come up with a change in the modules corresponding to CPU registers, pipelining, Cache memory and physical memory, which in turn turned out to be very cumbersome since M5 does not have a notion of shared memory, nor did it have any module which would help us abort a memory operation in the middle which turned out to make us too many changes in the source code of M5 simulator, which in turn crashed the simulator.