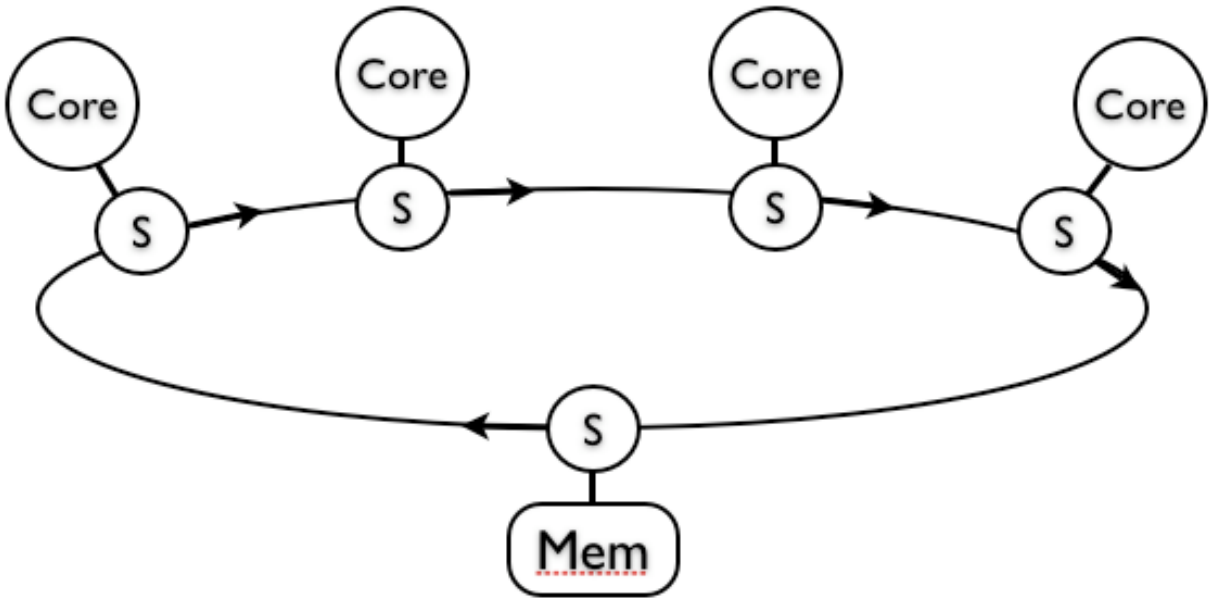


Vigilance

SMIPSv2 Scalable Multi-Core Processor with Ring Network and Direct Messaging Support.

Christopher Celio & Forrest Green
Massachusetts Institute of Technology
6.375: Complex Digital Design Final Project Report
14 May 2009

Abstract

We implemented a scalable SMIPSv2 multi-core processor. Scalability was achieved by implementing a ring network. Inter-core communication was achieved by providing every core a memory-mapped scratchpad memory that is accessible from any core. Using the scratchpad memory, software can be written using a direct messaging memory model. Benchmarks of matrix multiply, 1D Jacobi relaxation, and merge-sort are demonstrated and show an improvement in using four cores over a single core.

1 Summary

A multi-core processor was implemented by connecting several SMIPS cores into a ring network. Each core contains a single-issue, three-stage pipelined, in-order SMIPS processing engine, with out of order memory support and tournament branch prediction. Each core also contains an instruction cache, a data cache, and a scratchpad memory. Main Memory is provided its own network node, and all main memory accesses travel through the ring interconnect.

A number of single and multi-core benchmarks to evaluate the performance improvements have been implemented. We successfully synthesized two and four core versions on an Altera DE2-70 FPGA and had six and eight core versions working completely in simulation. Our only limit was the number of available logic slices on the FPGA.

An existing processor implementation from lab #3 in this course served as a starting point. A small local scratchpad was added to the cores and the instruction set was expanded to include a compare and swap instruction to facilitate inter-core communication. Hardware multiply was implemented to extend the usefulness of the processors and improve performance on multi-core message passing (which required multiply to do inter-processor communication). Cache coherence was not implemented in hardware for main memory and instead the software relies on the scratchpads (which are not cached) to handle communication or "flush" values into other cores.

We use a spiral development model in which we first worked on getting multiple core running while connected to the avalon bus and progressively added the desired features and migrated to the ring network.

2 Background

Presented below are some details about previous work and how it related to our project.

2.1 Background: SMIPS ISA

The SMIPS ISA (Smaller MIPS) is a subset of the MIPS ISA. SMIPS is a RISC (reduced instruction set computer) ISA. SMIPS lacks floating point support, misaligned load/stores, trap instructions, branch and link instructions, and branch-likely instructions, and other similar instructions.

There exist three versions of SMIPS. SMIPSv1 contains five instructions and is useful for in class discussion and testing of a processor's surrounding architecture. SMIPSv2 implements a wide variety of instructions, but lacks exceptions, interrupts, multiply/divide, linked load/conditional stores, and non-word aligned accesses. SMIPSv3 implements all of these, in addition to special instructions such as SYSCALL, BREAK, and SYNC.

For this project we will used the SMIPSv2 ISA, however, we will add the SMIPSv3 multiply instructions and implement our own Compare & Swap instruction.

The 6.375 course staff provide a smips-gcc tool-chain. We were able to run software written in both SMIPS assembly and C. Care had to be taken to not write code that could generate instructions not implemented by our processor (for example non-word aligned memory operations). Unfortunately we did not have access to the assembler code. This had the consequence that when we wanted to add new instructions to the ISA, we had to steal unused SMIPSv3 instructions.

Also, crt1.0, a common file to link to an application which handles the start up of the processor, had to be modified for our use to provide each core with its own stack space.

More Information regarding the SMIPS ISA can be found at:

(http://csg.csail.mit.edu/6.375/handouts/other/smips-spec.pdf)

2.2 Background: Lab 3 Processor

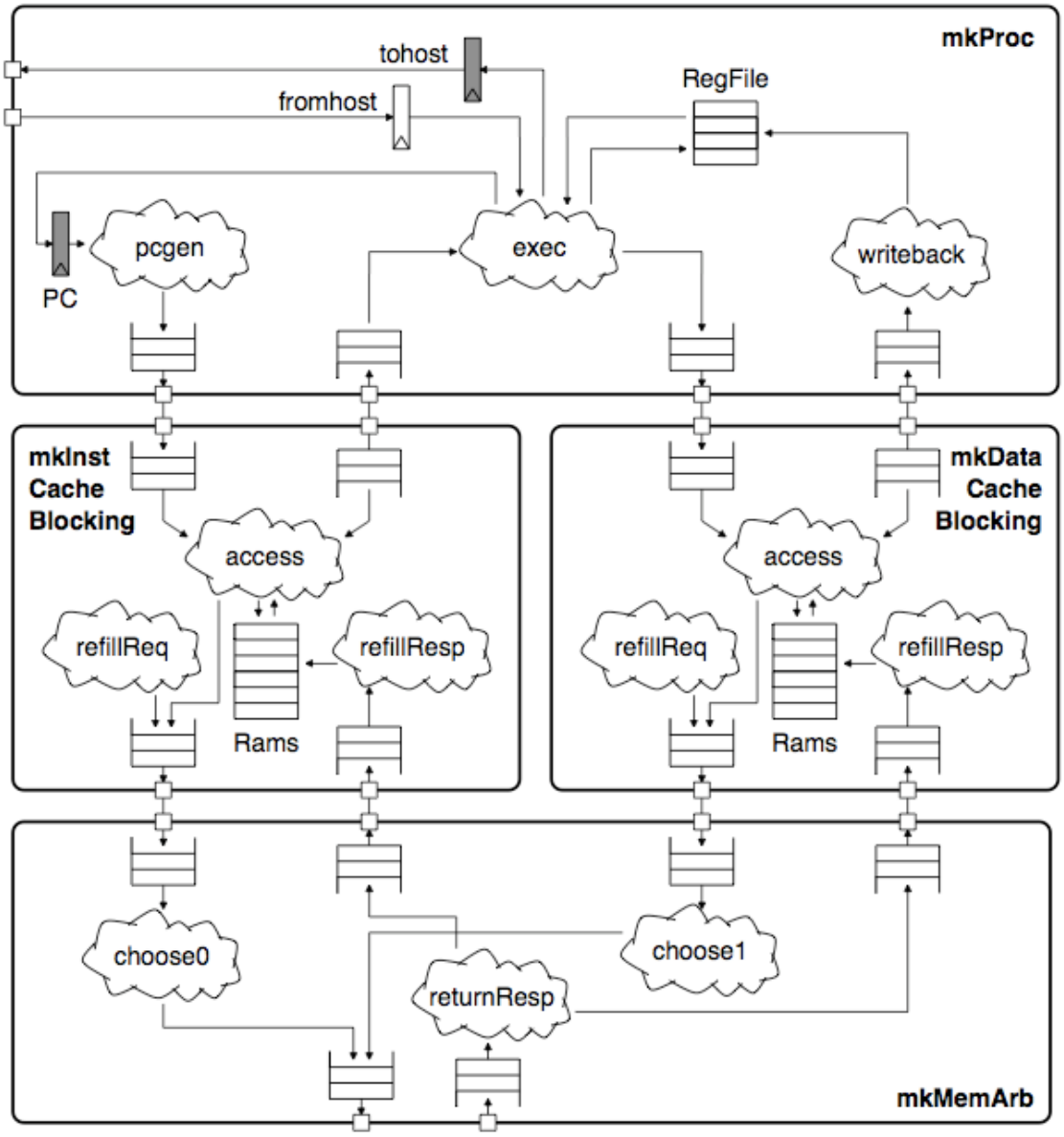


Figure: Cloud block diagram for SMIPSv2 multi-cycle (unpipelined) process, complete with blocking caches and arbitration to main memory. This processor was provided by the course staff as part of Lab#2. Image from the 6.375 Lab#2 handout.

We started with the SMIPS core we developed in Lab 2 and Lab 3. Its processing engine is a single-issue, 3-stage pipelined, in-order processor. As part of the course 6.375, students start with a single stage processor and build it into a three-stage pipelined processor with

branch prediction. A complete memory system is also provided, which include instruction and data caches, and a connection to an Altera Avalon Bus. "Main Memory" is then connected to the Avalon Bus, completing the system.

2.3 Background: Interconnect Networks

A number of different strategies exist in connecting multiply cores both together and to main memory. Crossbars and buses are commonly used in contemporary multicore processors. While we implemented crossbars for our early implementations - mostly used for testing the proof of concept and our software - we wanted to explore more interesting, and more scalable networks.

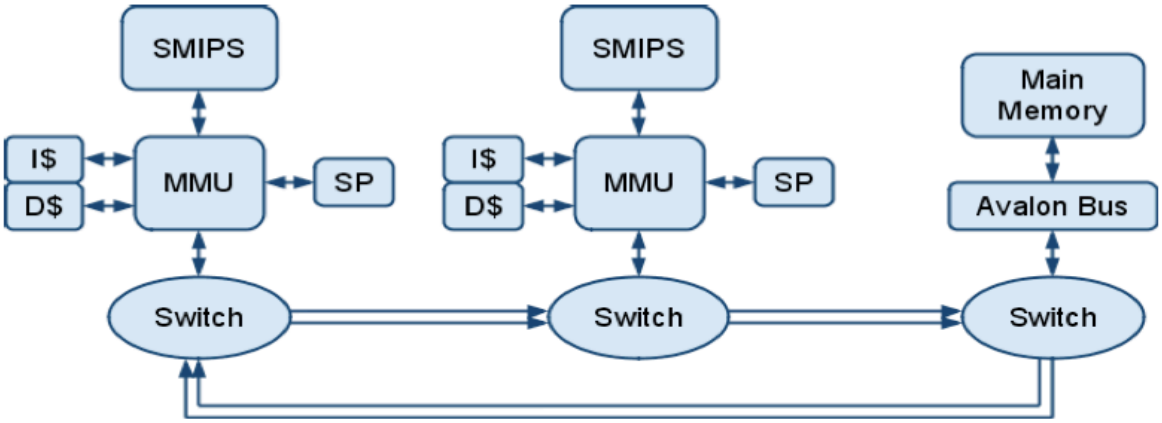
Using many cores, each containing a processing engine, caches, and a network switch, it is possible to connect cores in many different topologies which can vary widely in scalability, complexity, and performance. Point to point networks such as ring networks, mesh networks, toruses, and hypercubes can be realized. The ring network was chosen for this project for many reasons. It is scalable and can easily handle the small number of cores we hoped to synthesize (roughly eight). The logic is simple and can be quickly implemented. Simple is also convenience, because it also takes up a small amount of area and can be clocked very fast. For these reasons we see ring networks in commercial processors such as the IBM Cell Processor and Intel's Larrabee CPU/GPU hybrid.

2.4 Background: Communication with Shared Memory vs. Direct Messaging

There are two opposing models of programming parallel applications: shared memory and direct messaging. Using a shared memory model, all cores can see the entire memory space. Any modifications made to memory can be seen and accessed by other cores. This is a relatively easy and intuitive memory model to use for programmers, and it is reflected in common libraries such as POSIX threads. However, providing a shared memory model is very difficult for hardware. Hardware architects perform many optimizations, such as caching, that make it challenging to provide a coherent memory space. For example, a memory write by a core will be absorbed by its cache. If other cores wish to see the effects of that memory operation, some form of cache coherence will need to be provided. However, cache coherence is a complicated and demanding feature to provide, and we felt it was beyond the scope of this project. Therefore, we chose to instead provide direct messaging support.

With direct messaging support, each core has access to private memory. If a core wants to communicate with the other cores, it sends an explicit, direct message containing the data to be shared. To provide this support, we added a scratchpad memory to every processing node. Every core can read and write to any scratchpad. The scratchpads are memory-mapped. If core A needs to communicate with core B, core A can store a message to core B's scratchpad. When core B is ready to receive the message, it simply reads its own scratchpad.

3 Vigilance Processing



The following subsections contain details of the individual modules in each node of the Vigilance Processor.

3.1 Vigilance Processor: Smips Processing Engine

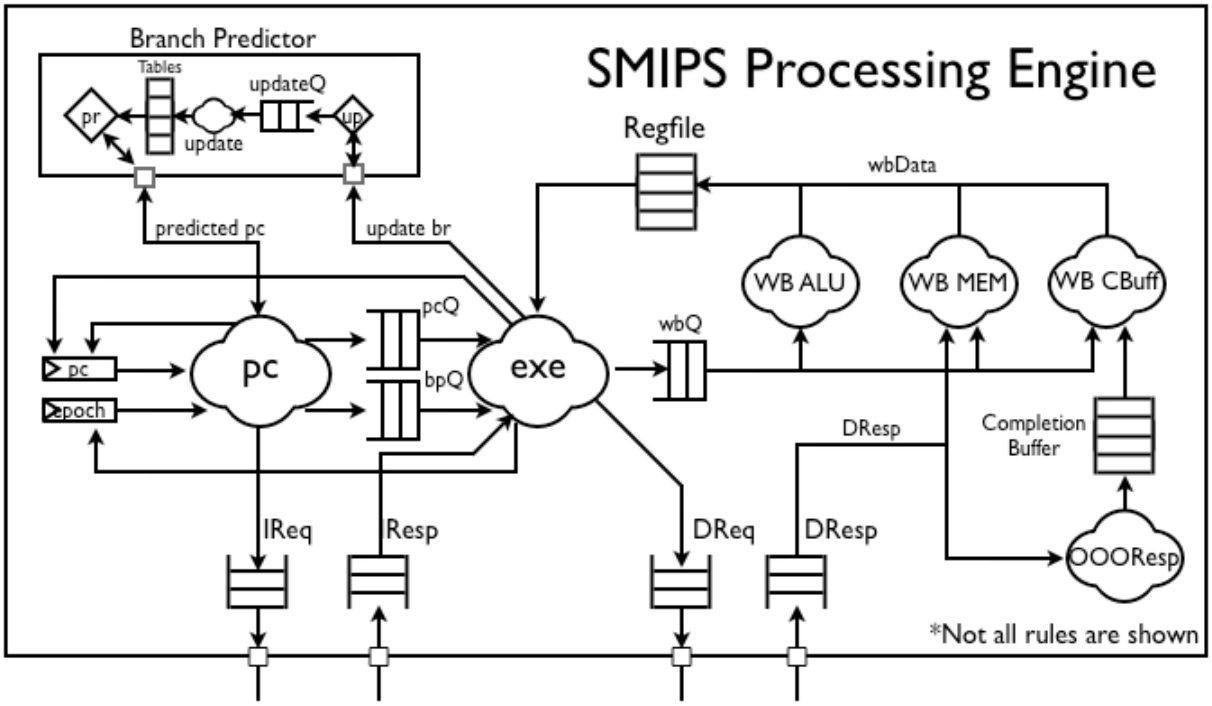


Figure: The SMIPS Processing Engine. It is a single-issue, in-order engine with out of order (OOO) memory support. It has three pipeline stages: instruction fetch (PC), execute (EXE), and writeback. Data responses received out of order are added to the completion buffer (OOOResp), and are written back when the issue tag at the head of the writeback queue (wbQ) matches a response in the completion buffer (WB CBuff). The processing engine also supports a pipelined tournament branch predictor.

The Vigilance processing engine is a single-issue, in-order, three-stage pipelined engine with out-of-order memory support. It is modified from the processor used in Lab#2 and Lab#3. It features an instruction fetch stage, an execute stage, and a writeback stage. The IF stage uses a tournament branch predictor to guess the next pc value. The branch is resolved in the EXE stage. The pc uses an epoch tag to mark instructions. The epoch is incremented on every branch mispredict. Then, instructions from out-of-date epochs can be discarded. The predicted branch is based from the IF stage to the EXE stage so that the branch predictor can be updated.

The EXE stage handles the central execution of the processing engine. The EXE stage also issues data requests, which are not resolved in the same cycle.

Any register file writes are sent through the writeback queue (wbQ) to the writeback stage. The EXE stage stalls if there is a register read conflict on registers remaining to be written back. Support for out-of-ordering memory operations also necessitate checking the memory address location as well. The register file is a "bypass regfile", which means that values being written to a register file can be forwarded to the readport of the register file at the beginning of the EXE stage's execution.

The writeback stage has been modified to handle out of order responses. This includes the addition of a completion buffer. See the "Out of Order Memory Operations" section for more detail.

3.2 Vigilance Processor: Out of Order Memory Operations

Out of order memory operations can occur due to two different features in our processor. The first is due to packet bouncing. It is thus possible for two operations to be issued, but the first request can be bounced if it reaches the memory node when it's pipeline is full. In such a situation, the second request can reach the memory and be serviced first. Likewise, memory responses can also be received by the core out of order.

The second cause of out of order memory operations is due to the existence of multiple memories that can service memory requests. A processor can issue a request to main memory, its local scratchpad, and a remote scratchpad all at once. Because each is located in different locations on the network, and may have different loads, we can expect each memory to exhibit different access latencies. Therefore it is reasonable to expect that the responses from each can be received out of order.

Two modifications to the processing engine were required to provide support for out of order memory operations. The first is to verify that all issued requests do not depend on outstanding memory requests. The writeback queue was expanded to include not only the writeback register destination, but also the memory operation address. New memory operations would have to touch a different address, otherwise, the execution stage would stall the memory operation until the offending instruction is written back and retired.

The second modification to the processing engine involved re-ordering responses and

writing them back and retiring in-order. This was accomplished by adding a timestamp, or "issued" tag, to every writeback queue item. This "issue" tag was also added to the memory request/response header so returning items could be correctly ordered. If the returning memory response did not match the issue tag at the front of the writeback queue, the response was written to a "completion buffer", indexed by issue tag (See Figure Smips Processing Engine). If the issue tag at the front of the writeback queue matched a memory response in the completion buffer, then the item in the completion buffer was written back to the register file and removed from the completion buffer.

3.3 Vigilance Processor: Branch Prediction

The processing engine utilizes a tournament branch predictor to improve IPC. The predictor is modeled after the Alpha 21264 predictor. It has three internal predictors: a global history predictor, a local history predictor, and an arbiter history predictor. The global history contains a register that tracks the history of all branches (taken vs. not-taken) and indexes a bimodal counter table based on the global history. The local history predictor contains an array of local histories indexed by pc. The local history, found by pc, is then used to index a bimodal counter table to predict taken/not-taken. An arbiter bimodal table is indexed by pc to predict whether the global history or local history prediction should be used. A target table, indexed by pc value, stores previous "next pc" values, based on updates provided by the Execution stage. If the predictor decides a branch is to be taken, the predictor submits to the PC Generation rule a "target address" that it predicts is the next pc value.

In the Lab#1 Verilog processor system, in which the processor is hooked up to single cycle, "magic memory", the tournament predictor garnered an average 95.86% IPC (100% is a maximum 1 IPC) across the five benchmarks median, qsort, towers, vvadd, and multiply. The best performance was a 99.4% on vvadd. , while the worst performance was 92.94% on median. For comparision, predicting a constant pc+4 averages 84.88% with a best of 90.76% and a worst of 77.77%. Performance improvements are not as pronounced in the Vigilance processor since a full memory system exists. Therefore, substainially lower IPCs exist due to cache misses and long memory latencies.

3.4 Vigilance Processor: Caches

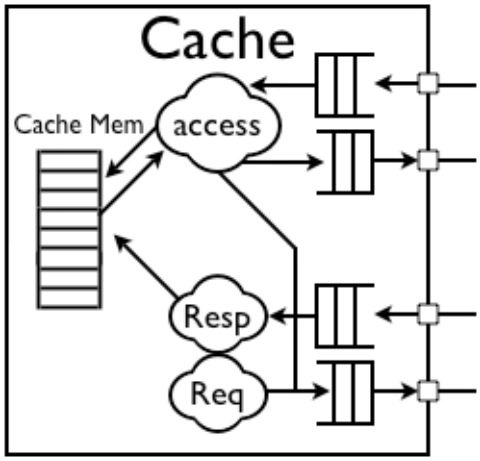
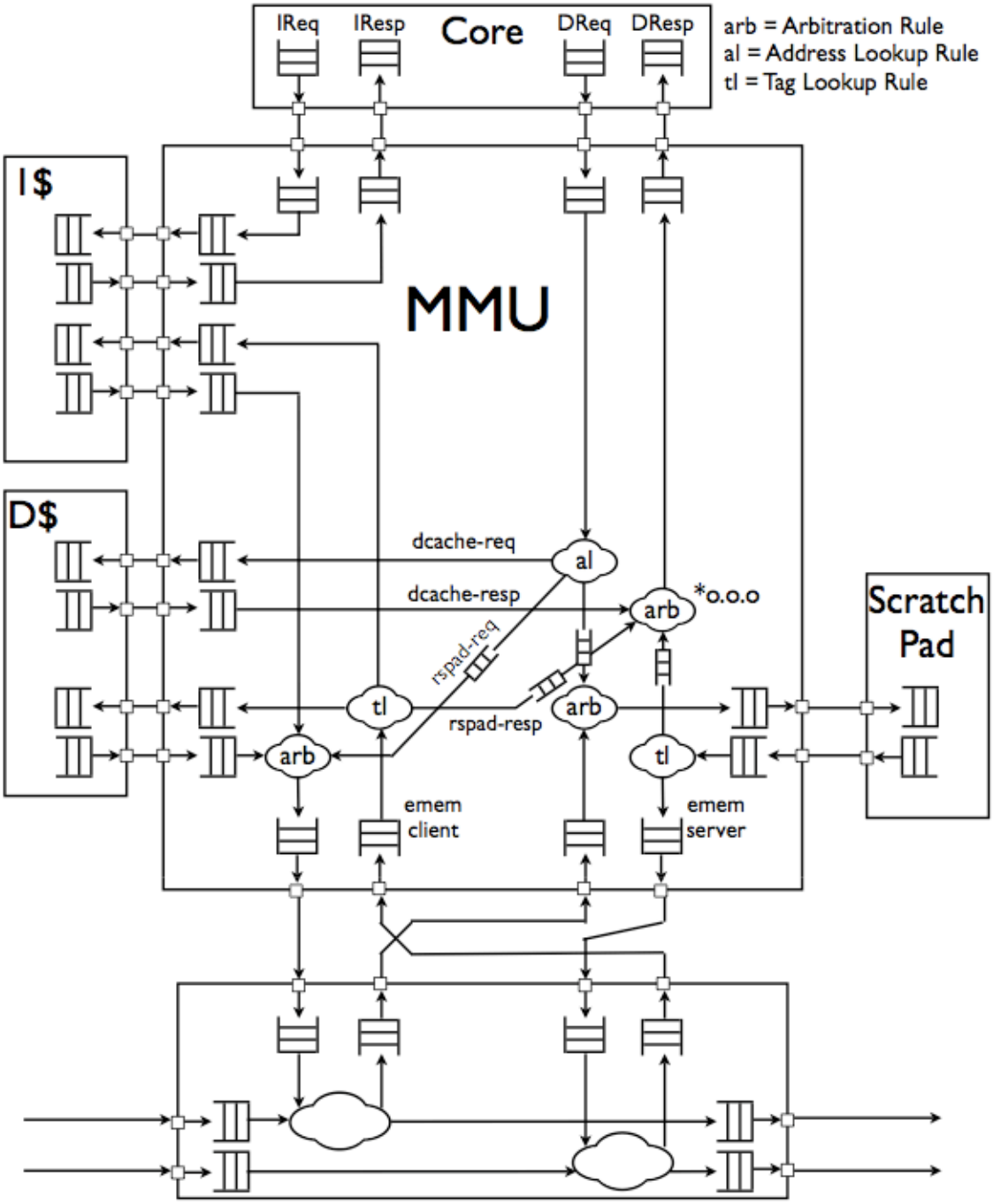


Figure: Block Diagram of a Blocking Cache.

The caches used in this project were provided by course staff as part of Lab#2 and Lab#3. They are simple, blocking caches that allow for only a single outstanding memory request to main memory on a refill. The caches are direct mapped and hold only one word per line (32 bits). They do not have a dirty bit, and thus writeback all evictions.

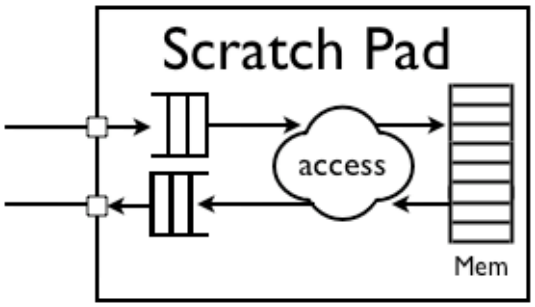
Early in the project, we modified the caches to allow all memory requests to pass through. This was very useful, as it allowed us to test multicore applications in a shared memory environment (no cache coherence problem to deal with).

3.5 Vigilance Processor: Memory Management Unit



The MMU had perhaps the most complicated interface of any part of the project. It connected to instruction and data request ports on the processor, the two caches, the scratchpad, and the network switch. It contained several arbiters that would route requests between different request and response queues. There is one place (labeled on the diagram as O.O.O.) where responses can actually get out of order. Rather than trying to add additional support to ensure that these stay in order, reordering is done in the processor which is already necessary to handle reordering in the ring network.

3.6 Vigilance Processor: Scratchpads



In order to provide direct messaging support, each core is provided a memory-mapped scratchpad memory. Every core can read and write to any scratchpad. Send and Receive functions can be written in software, using the functionality provided by the scratchpad processors. Special support was added to the scratchpad to handle Compare and Swap requests and ensure that they were atomic. Because the memory in the scratchpad was in a register file it was possible to handle Compare and Swap requests in a single cycle and avoid needing any additional state as in the bus controller.

3.7 Network Switches

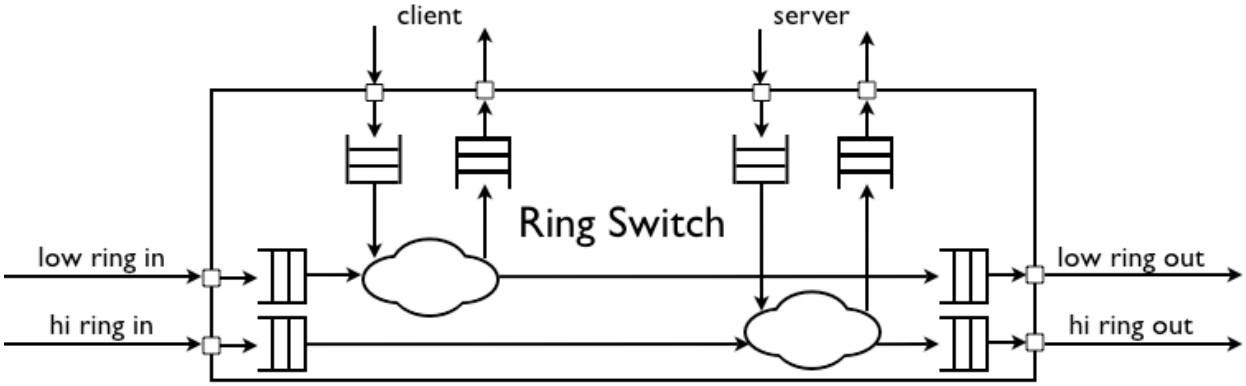


Figure Ring Switch. The ring switch contains logic for both the low and hi priority rings. The processing core sends its requests through the client interface and receives its responses through the server interface. The scratchpad response/requests, as well as the memory node connections, receive responses on the client interface and send their responses through the server interface.

Each node has its own network switch which routes memory operations between the MMU and the adjacent network switches. Switches were responsible for checking if incoming packets were addressed to them and ensuring that there was space to enqueue additional outgoing packets. To avoid inserting bubbles into the network there was some additional logic to ensure that as long as an incoming packet was for this processor, the outgoing packet could immediately be enqueued.

3.8 Vigilance Processor: Cache Coherence

The Vigilance Cache Coherence algorithm is very simple: programmers be vigilant, your cache is not coherent. Correctness is ensured by not caching any of the scratchpads. To pass information between processors it is necessary to write that information into the scratchpads. This is not as convenient for the programmer as a full shared memory model, but it is much simpler than attempting to manage cache coherence. It would be a fairly small modification to make volatile read and write instructions so that main memory could also be used for passing messages or for easily condensing data to a central location after distributed computations. (In theory this could already be done by passing the value to each processor and having it write it to main memory which would ensure that the correct value was either in the cache of each processor or main memory, but this would be fairly inefficient).

3.9 Vigilance Processor: Extensions to ISA

We added a few extensions to the ISA including a compare and swap instruction, a memory fence instruction, and six instructions related to multiplication.

There is no native compare and swap instruction in any version of the SMIPS ISA so we choose to use the conditional store instruction so that the assembler would accept the opcode. Compare and swap also required some additional support in the bus controller and the scratchpads. Loads and stores in main memory take multiple cycles so some additional state was added so that no intermediate memory requests could be issued that would interfere. Currently no requests of any kind are allowed while a CAS is pending which might create some dead cycles.

The memory fence instruction ensure that responses have been received for all of the memory operations that were previously issued. This is important for some situations where memory operations on different operations and different address would actually be dependent on each other. For example, imagine if one processor stores a value in a shared variable and then releases the lock. Neither operation has explicit dependencies so they could both be issued, however if they were absorbed by memory in the wrong order another processor could see that the lock is released and read the value of the shared variable before it is updated by the other variable. Issuing a memory fence request just before the lock is released will ensure that this cannot happen.

3.10 Vigilance Processor: Bus Controller

The Bus controller existed as its own node on the ring network. Its primary job was to convert memory operation requests to interface with the avalon bus and ensure that the responses would be routed back to the correct processor with appropriate tags from the request.

3.11 Vigilance Processor, Bus Controller: UART Simulation Support

It is fairly easy to introduce bugs in a project of this sort with fairly subtle effects. When a benchmark gets the wrong result it can be very difficult to figure out where the error is

occurring, even when the full execution trace is available. In these situations printf can be a very valuable tool (assuming it doesn't break the behavior). Because recompiling the hardware for the FPGA can take a very long time, we added support in the bus controller so that we could use serial output in simulation. It would intercept the UART address and provide appropriate status responses or make $display calls that were later extracted from the output of simulation by a script to reconstruct the serial output. Although the timing on this (as well as the substantial time for printf calls) could affect behavior, we found that it was still a valuable tool for debugging.

3.12 Vigilance Processor, Bus Controller: Compare and Swap Support

In order to ensure that the compare and swap instruction executed atomically, support was added to the bus controller. This was done by adding a small amount of state that would avoid issuing any other requests while a CAS load was pending and automatically issue the second store instruction based on the match. In theory it would be possible to issue other requests while the CAS was taking place so long as the same address was not affected, but we decided to accept the small potential performance penalty to avoid complicating the hardware.

4 Vigilance API

We implemented a set of C routines that were shared between the benchmarks and test applications.
To allow for quick authoring of software benchmarks, an API that encompassed both commonly used code and access to the underlying hardware features was created. For example, the API provided access to locks and synchronization primitives, direct messaging support, memory fencing.

4.1 Vigilance API: Peterson Software Locking

Locking support was initially provided through a Peterson's software implementation for two cores. Our implementation is shown below:

```
void acquirePetersonLock(volatile VGLock* lock, int thread_id)
{
  if ( thread == 0)
  {
    lock->flag0 = 1;
    lock->turn  = 1;

    while( lock->flag1 == 1 && lock->turn == 1 )
    { }; //wait
  }
  else
  {
    lock->flag1 = 1;
    lock->turn  = 0;

    while( lock->flag0 == 1 && lock->turn == 0)
    { }; //wait
  }
}
```

4.2 Vigilance API: Compare & Swap Hardware Locking

With the need for a lock that can support more than two cores, we chose to implement a hardware lock. Because we did not have access to the assembler code, we used an existing SMIPSv3 instruction for use as a CAS. Coincidently, stored-conditional was used as its form matched our CAS instruction.

4.3 Vigilance API: Barriers

At a barrier each core waits for a completion signal from the core before it and then passes it on (one can imagine a token being passed around the ring). The first core immediately sends a completion signal and the last core starts the chain again with a resume signal that causes cores to reset their flag and resume execution. There is a small bit of additional logic which ensures that a completion flag is not cleared at the end of a barrier which would could happen if the parent core finished the barrier and started a second barrier before the child core finished resetting the flag.

4.4 Vigilance API: printf

We ported a version of printf used in the standard c library for JOS. After some modifications to use our hardware divide rutine we had support for things like formatting output and printing in different bases (although still no floating point).

4.5 Vigilance API: Stats

We created two seperate performance counters that could be activated in software to count executed instructions and total cycles for various sections of code. By using multiple

counters we were able to count the total time taken for a benchmark and look at what part of that time was spent in specific sections of the code.

4.6 Vigilance API: Multiply/Divide/Modulus

We implemented software multiply, divide, and modulus functions. These were very useful for things like computing the address of a neighbor's scratchpad in things like direct messaging and were also good for creating compute heavy benchmarks. We found that for tasks like direct messaging a large fraction of the time was being used to compute the address of other scratchpads. Originally our processor did not support the hardware multiply instruction from the SMIPS ISA. However, when we noticed that barriers spent 800 of the total 1000 cycles performing address calculations, we decided to implement hardware multiply.

4.7 Vigilance API: Memory Fence/Compare and Swap

Because the c language doesn't have any native method for requesting memory locks or using compare and swap instructions we implemented small wrappers to make these easier to use at the c level although in the end these were rarely used outside of the API.

4.8 Vigilance API: Blocking Send/Recv

Send and receive was done using reserved sections of the scratchpad memory. Care was taken to ensure that loops would not issue rapid requests for the remote scratchpad and generate excessive network traffic.

```
//Multiple copies of these flags exist for each possible sender or
receiver
volatile int permission; //sender flag indicating that the receiver is
ready
volatile int intent; //receiver flag indicating state of the sender
volatile int value; //receiver flag holding the value to be sent
send(int new_value) {
    intent = READY_TO_SEND; //indicate that a value is ready to be sent
    while(permission != ALLOWED_TO_SEND) {} //Wait
    value = new_value;
    permission = NOT_ALLOWED_TO_SEND;
    memory_fence(); //ensure that the new_value is correctly written
    intent = SENT;
}
int receive() {
    while(intent != READY_TO_SEND) {} //Wait
    permission = ALLOWED_TO_SEND;
    while(intent != SENT) {}//Wait
    return value;
}
```

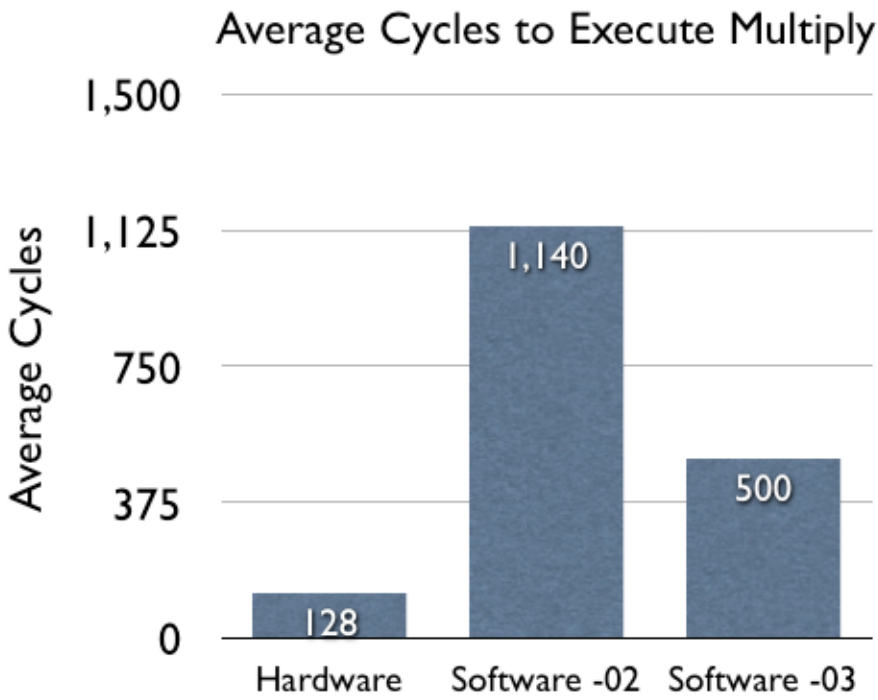
5 Benchmarks

In addition to many single core tests we implemented a number of benchmarks with different sharing patterns and compute to memory request ratios. Below is provided the memory parameters we used.

Inst Cache	256 bytes (64 words)
Data Cache	1024 bytes (256 words)
Scratchpad Memory	128 bytes (32 words)

The instruction cache is very small, however, the benchmarks we run are also very small. The scratchpad is only used for API specific operations, and therefore does not need to be very large. The data cache is capable of holding 256 words of information. For many of the following benchmarks, we sweep the problem size to show performance for the following conditions: the problem fits entirely within a single core, the problem fits entirely within the space of the sum of all onchip caches, and the problem does not fit within the caches. Smaller core sizes are at a slight disadvantage in these benchmarks because we did not resythesize the hardware for the different numbers of cores and merely disabled the extra cores in software. This means that there would be an increased latency in accessing main memory over a truly smaller number of cores (or one connected directly to memory).

5.1 Benchmarks: Multiply/Memory Benchmark

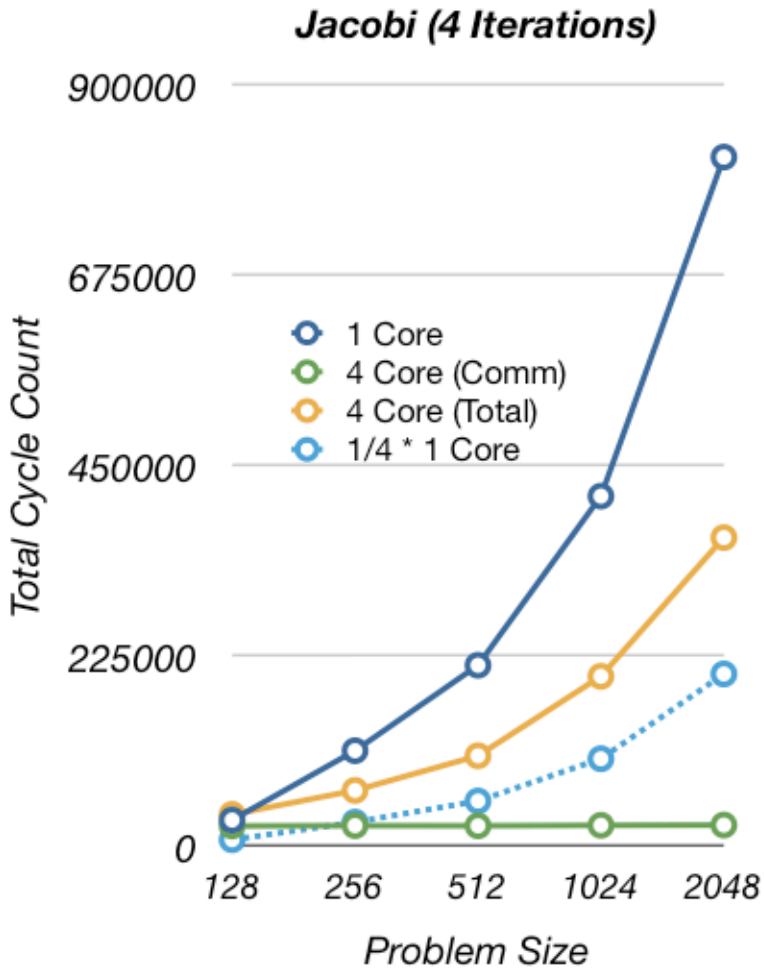


5.2 Benchmarks: 1D Jacobi Relaxation

1D jacobi relaxation is a algorithm with a nearest neighbors sharing pattern. The psuedocode for a single-core is as follows:

```
for (every iteration) {
  for (entire array) {
    array[i] = (oldarray[i-1] + oldarray[i+1]) /2;
  }
  swap(array, oldarray);
}
```

In the parallel version, each core is given a contiguous piece of a 1D array. Sharing only occurs at the boundaries at each array. Therefore, at the end of each iteration, each core sends the leftmost and rightmost elements to his neighbors using the direct messaging API.

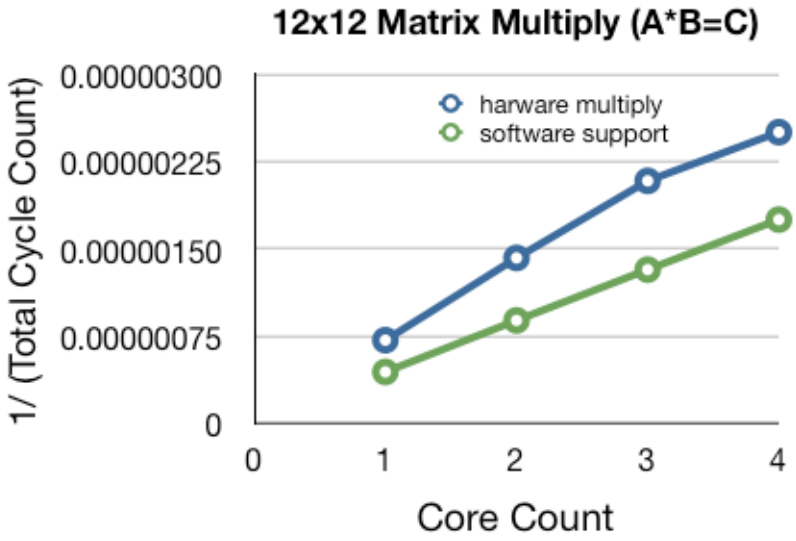


In the above chart, we are comparing single-core and four-core run times. We show the

effect of increasing problem size and how the advantage of multiply cores has increasing benefits as the problem size increases. We also plot the number of cycles spent communicating between the cores. It is a flat line, because the communication overhead does not change with problem size (cores still only need to message the leftmost and rightmost elements of their arrays). With a small problem size, we see that communication overhead overwhelmingly dominates application run time.

We also plotted the "1/4th single-core run time" to provide a visual mark for where a 4x speed over single-core runtimes up would lie. Instead, we see that four cores is only twice as fast as the single core runtime.

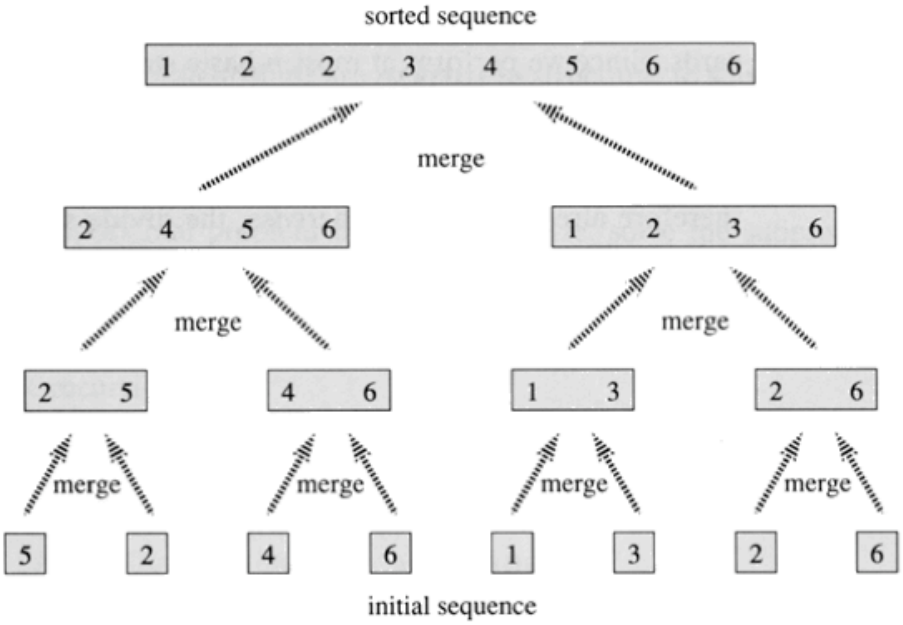
5.3 Benchmarks: Matrix Multiplication



We implemented a simple distributed matrix multiply where each core was responsible for different columns of the output matrix. The computation was done simply by taking full dot products. This did not attempt to take advantage of ring network to pass portions of the loaded matrix around nor did it try to avoid evicting values from the data caches. We ran versions of this benchmark using hardware and software multiplies with 1 through 4 cores actively participating. Using the software multiply this benchmark had an almost exactly linear speedup as more cores were applied to the computation. The hardware multiply speedup is not quite linear. This is probably the effect of the higher ratio of memory requests to local operations which results in cores competing for main memory bandwidth. Given more time it would have been nice to add instrumentation to the bus controller to directly measure how close we were able to get to saturating the memory bandwidth.

5.4 Benchmarks: Merge-sort

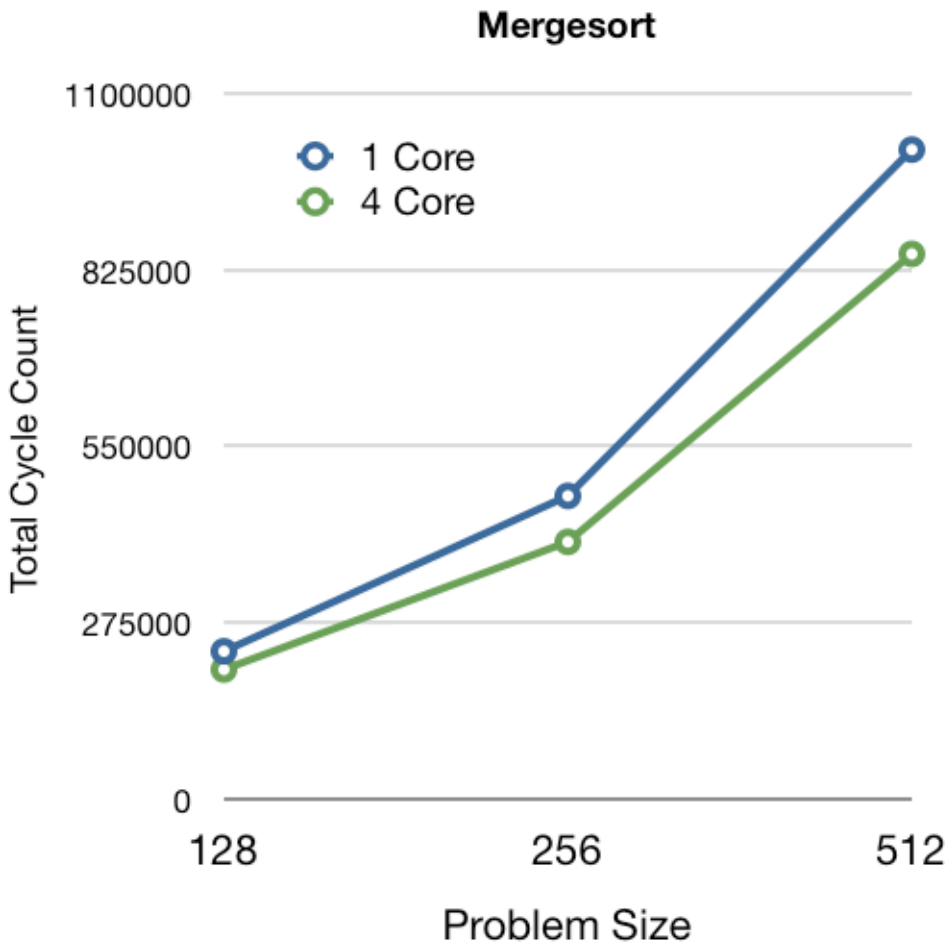
Merge-sort works by breaking down an array recursively into its smallest parts, and then merging the smaller, sorted pieces until it has recursively sorted the entire array.



http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm

This divide and conquer strategy means we can easily give a piece of the unsorted array to each core. Then, each core sorts its own piece of the array in the normal, sequential fashion. The only difference is that once each core has sorted its array, the algorithm needs to perform a parallel merge on the sorted arrays.

In the interests of time, our final merge step is implemented in a sub-optimal fashion. We have the cores send their elements of the arrays to core #0. Core #0 then performs the final merge step sequentially. A more optimal solution would be to merge the sorted arrays in a tree-like communication fashion. The cores send their sorted data to a neighbor core, who merges the data, who then continues the pattern. For two cores, core#1 simply sends his information to core#0, who completes the algorithm with a final sort. For four cores, core#1 sends data to core#0 (who then sorts), and core#3 sends data to core#2 (who then sorts). Once this step has occurred, core#2 sends the sorted array (coalesced from core#2 and core#3) to core#0, then performs the final sort.



Despite our sub-optimal implementation of the final merge step (all of the work is performed by core #0), we still find that it is more advantageous for all tested data points to use four cores instead of one. This is somewhat suprising, because the problem size shown here are fairly small.

6 Further Work & Project Extensions

The main goal of this project was to demonstrate a bluespec implementation of a ring-based processor with direct messaging support synthesized and running on an FPGA. While we were successful in this endeavor, much work still remains.

First, it would be interesting to write more intensive and varied applications that flex different patterns of sharing and partitioning of data. Some models include pipelining (instruction partitioning) and work queues. It would also be nice to collect more benchmark data that explore main memory saturation, ring network utilization, cache miss rates, and processing engine utilization. We also had issues with fitting four cores on the FPGA. In an attempt to squeeze four cores (which took up 93% of logic slices), timing constraints were violated in one of the cores' completion buffer writeback paths. We would like to be

able to explore and possibly optimize this data path. Indeed, we would like to further explore bluespec scheduling optimization across the entire processor.

In addition to this work, there were many small optimizations and interesting extensions that we thought of while working on the project, unfortunately we did not have time to include all of them. Here is a selection of some of the more interesting experiments and features that we would have liked to perform.

6.1 Possible Project Extensions: Volatile load/store to main memory

Because all communication is done core to core, it is actually a fairly substantial task to collect all of the results in one place. It would be a fairly simple extension to add a volatile load/store request so that once a distributed computation has been completed the results can be directly written to main memory.

6.2 Possible Project Extensions: FIFO depth tuning

The focus of the project was primarily on achieving full functionality on multiple cores. That we were actually able to measure a non-trivial performance improvement was an incidental (though expected) benefit. We suspect that a significantly higher ipc could be achieved simply by fine tuning the depth of the fifos in the processors. We did not change the values inside the processing engine which were intended to handle a latency of a few cycles at most. We also defaulted most of the fifos in the mmu and switches to either size 2 or bypass fifos. Careful analyisis and tuning of these could easily increase the performance of the processor.

6.3 Possible Project Extensions: FPGA Exploration

For this project, the FPGA was underutilized, as all our software could run perfectly well in simulation. Also, simulation provided a much smaller feedback loop to providing hardware testing and tweaking. However, we wished we could explore more FPGA-specific features such as VGA support and terminal support. Having video-out would better motivate providing more optimized hardware, and provide a great incentive to putting more cores on an FPGA.

6.4 Possible Project Extensions: Nix the Nios

Having added full support for printf and debugged the processor it would be reasonable to completely remove the Nios coproccessor and make room for more cores on the FPGA.

6.5 Possible Project Extensions: Main memory CAS optimization

Currently no requests of any kind are allowed while a CAS is pending which might create some dead cycles. It would be a fairly small change to simply check that no additional operations are added for the CAS address or even creating a full scoreboard style module so that even multiple CAS operations could be pending simultaniously.

6.6 Possible Project Extensions: Arbiter optimization

There are a number of arbiters, particularly in the MMU, that have uniform priorities. Careful analysis of these might make it possible to prioritize things like scratchpad responses or main memory data loads to slightly improve performance.

6.7 Possible Project Extensions: Bounce Delay

When a memory operation is bounced around the network it must complete a full additional cycle. While this is necessary to avoid deadlock, it introduces a large delay which would most likely be avoided by stalling for just one more cycle. For 2 and even 4 cores this is probably uncommon, but we have not measured this and for example in the matrix multiply benchmark we can already see effects from competition for memory bandwidth.

6.8 Possible Project Extensions: Scratchpad aware software fifos

For the current API when sending a direct message the sender and receiver block. We sketched out, but did not have time to implement an algorithm for creating a FIFO that would only need to block when it was full by keeping separate generated and consumed counters in the scratchpads and storing values in a ring buffer. Currently, sending a message is a comparativly expensive process that only sends one word. By using FIFOs on either end it would help avoid contention and allow more effecient use of the processors especially in the absence of an OS that could schedule other threads while a core was waiting.

6.9 Possible Project Extensions: Memory request segmentation.

Network packets encapsulated entire memory request and responses, however, this forces the ring to have very wide data paths parts of which are unused for the smaller packets. An interesting optimization, seen in commercial processors, would be to divide memory operations into multiple, small packets. This would then require the network switches to be able to divide traffic, or macropackets, into small packets, and be able to reassemble small packets into a single memory request/response. Many issues would arise, such as how to deal with packet bouncing, saturating the ring before all of the packets for a macropacket could be added, and interleaving of packets between different macropackets.

7 Interesting Bugs & Obstacles

In the interest of future projects, we would like to collect some of the bugs we encountered. Bugs encountered included: cores sharing the same stack space, failing to stall the execute stage for some instructions, incorrectly tagging memory requests or losing tags as they go through the memory system causing incorrect routing, not handling all cases for case/tagged statements, and benchmark bugs where unnecessary work was being done. Dealing with the outside build tools also brought in many errors, such as writing a verilog wrapper that mishandled address bits.

8 Conclusions

Multi-core programming, especially in the absence of cache coherency, is difficult and requires careful attention to locking, atomicity, data sharing, correctness, reordering, and performance. Despite this, we were still able to implement a number of benchmarks. Overall we consider this project very successful. We achieved our primary goal of distributed software running on multiple cores in a ring network on the FPGA. Additionally we were able to simulate as many as 8 cores although this was too big to fit on the FPGA we were using. Even though performance was not a primary goal of this project we ended up seeing performance improvements in our benchmarks even for fairly small problem sizes.