ECE 4750 Computer Architecture, Fall 2013

Lab 5: Multicore Processor

School of Electrical and Computer Engineering
Cornell University

revision: 2013-12-03-09-41

In this lab, you will be composing the components you have designed throughout the year to create a multicore system composed of processors, networks and caches. You will get the chance to program scalar and parallel programs in C, and see the compiled and assembled code, and it running on your processor system.

You will work up to the final design by using an incremental design process by composing first a single core design and then the multicore processor. We will evaluate the completed multicore processor using the provided multithreaded benchmarks.

Unlike the earlier labs, this lab also has a software focus and you will write a scalar quicksort algorithm and a parallel sorting algorithm of your choosing and show the performance difference.

As with all lab assignments, the majority of your grade will be determined by the lab report. You should consult the course lab assignment assessment rubric for more information about the expectations for all lab assignments and how they will be assessed.

This lab is unique in its focus on structural composition of elements that you have already built and on software, This lab is designed to give you experience with:

- compose a single core processor system using processors and caches previously built
- compose a multicore system which is a significantly more complicated design
- write software for both a single threaded and multi-threaded programs

This handout assumes that you have read and understand the course tutorials. To get started, you should access a course development machine and perform the following steps:

```
% cd ${HOME}
% source setup-ece4750.sh
% ece4750-lab-admin start ece4750-lab5-mcore
```

Because some tests use compiled code, we also need to compile the assembly tests and the benchmarks. Note the `--host=maven` flag for `configure`. This uses the "maven" compiler installed in `amdpool`, which compiles for the PARC ISA.

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/apps
% cd tests
% mkdir build
% cd build/
% ../configure --host=maven
% make
% ../convert
% ../convert-cache
% cd ../../ubmark
% mkdir build
% cd build/
% ../configure --host=maven
% make
% ../convert
% ../convert-cache
```

Note that you might see a warning saying `mv: rename *.d to dep/*.d: No such file or directory`. It is safe to ignore this warning. Once you have the apps compiled, you can go on as usual:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore
% mkdir build
% cd build/
% ../configure
% make
% make check
```

The file structure for plab5-mcore/ directory looks similar to before:

- `plab5-mcore.mk` – makefile fragment
- `plab5-mcore-ProcCacheNetBase.v` – baseline single core composition
- `plab5-mcore-ProcCacheNetBase.t.v` – tests for the baseline single core composition
- `plab5-mcore-ProcCacheNetAlt.v` – alternative quad-core composition
- `plab5-mcore-ProcCacheNetAlt.t.v` – tests for the alternative quad-core composition
- `plab5-mcore-mem-net-adapters.v` – adapters to translate memory messages to network messages for the data cache banks and vice versa
- `plab5-mcore-mem-net-adapters.t.v` – tests for the memory network message adapters
- `plab5-mcore-MemNet.v` – memory request/response network
- `plab5-mcore-MemNet.t.v` – tests for memory request/response network
- `plab5-mcore-test-harness.v` – shared test harness for all of the implementations
- `plab5-mcore-test-cases-*.v` – test cases for different instructions, same as Lab 2
- `plab5-mcore-test-cases-vmh.v` – test cases using compiled self-checking assembly tests
- `plab5-mcore-sim-harness.v` – simulation harness for all of the implementations
- `plab5-mcore-sim-base.v` – simulator for the single core composition
- `plab5-mcore-sim-alt.v` – simulator for the quad-core composition

In addition, the `apps/` directory contains self-checking assembly tests and C source codes for the micro benchmarks. Self-checking assembly tests are a different approach to the testing strategy you have used in Lab 2. In Lab 2, you have used explicit test sources and sinks for inputs and outputs. The self-checking tests do testing in software and only notify the manager in case of a failure or success, as opposed to sending every single result back. The main drawback of using this approach is that it requires a lot more instructions to be working before testing can work. For instance, it requires the software be able to compare a result to an expected value, and branch to pass or fail labels in the code, and send this pass/fail information back to the manager. In comparison, using explicit sources and sinks in Lab 2 only required the `mtc0` and `mfc0` instructions working.

The self-checking assembly tests are in the `apps/tests/` directory. Some of the simpler instructions are tested in `parcv1/` and the others are in `parcv2/`. In addition to the single threaded assembly tests, there are also multi-threaded assembly tests in the `mt/` directory. Because we need the processor to run compiled code, it needs to implement more instructions than was required for Lab 2. To ensure we test the new instructions, we use these self-checking assembly tests. Take a look at how these tests are implemented. It would also be interesting to compare your additional tests you have implemented in Lab 2 to the self-checking versions. As shown before, you can compile these assembly tests by going to its build directory and running the following commands:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/apps/tests/build
% make
% ../convert
% ../convert-cache
```

To run src/sink based or self-checking assembly tests, we have different commands. You can try them in the Lab 2 processor. Running `make check-plab2-proc` or `make check-plab5-mcore` run the src/sink based assembly tests. Running `make check-vmh-plab2-proc` or `make check-vmh-plab5-mcore` run the self-checking tests:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/build
% make check-plab2-proc                  # runs src/sink for Lab 2
% make check-vmh-plab2-proc              # runs self-checking for Lab 2
% make check-plab5-mcore                 # runs src/sink for Lab 5
% make check-vmh-plab5-mcore             # runs self-checking for Lab 5
```

You should run both types of the assembly tests to make sure your design works.

The micro benchmarks are similarly in the `apps/ubmark/` directory. You can find the C codes and the dataset files in `apps/ubmark/ubmark`. You can compile it using the same approach:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/apps/ubmark/build
% make
% ../convert
% ../convert-cache
```

You can use the `ece4750-lab-admin` script and the `git` command to add members to your lab group, use your own group repository to collaborate with your group members, and ultimately prepare your submission for uploading to CMS. **Remember to use the `ece4750-lab-admin` script to create the submission tarball. All the tests, simulation experiments and VCD dump generation must be run in the build directory!**

1. Introduction

Ever since the transistor scaling have stopped delivering exponential growth in performance and shrinkage in area while still within a constant power budget, computer architects have tried different ways to scale the performance of processors into the future. Usually the idea is parallelism, and different processor designs exploit different amounts of data-level parallelism (DLP), instruction-level parallelism (ILP) or thread-level parallelism (TLP).

Multicore processors have been popular in the mainstream in the last decade. This design duplicates the processor core and enables different program threads to run at the same time on a different processor, exploiting TLP. Ideally, if you can parallelize an application to take advantage of four processors, you can get speedups very close to 4X.

However, multicore designs also have their drawbacks. First of all, it is not possible to fully parallelize most applications, and previously non-significant non-parallelizable parts of the code start to dominate in terms of execution time. In addition, to ensure correct execution, you will need to look into coherency, consistency and synchronization, most of which have a trade off of performance gains through aggressive mechanisms and ease of programming. Lastly, the multicore design has hardware complexity and more hardware overall to enable this scheme.

In this lab, you will get the chance to explore the trade offs of using a single core, relatively straightforward machine, and a quad-core system with a banked data cache, and private instruction caches. The multicore system is significantly more complicated, and you will have a chance to build your own quad-core system. Then you will analyze the performance gains, if any, that you get from each configuration running benchmarks, including one that you wrote.

2. Baseline design

For the baseline design, your task is to compose the bypassing processor from Lab 2 and two caches from Lab 3, one as the instruction cache, the other as the data cache. This will give you a simple yet realistic single-core system. Make sure your design passes both types of assembly tests by running:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/build
% make check-plab5-mcore
% make check-vmh-plab5-mcore
```

You will compose these components in `plab5-mcore-ProcCacheNetBase.v`. Note that despite the naming, this design does not need a network. You should enable line tracing that is currently commented out in this file so you can see what is going on in your single core system.

Both the cache and the processor had some modifications to be able to compose them and run C programs on them. The processor now supports more instructions, which are required from compiled programs. The processor also supports a special "stats" bit to tell the manager that stats should be enabled. The reason we need this stats bit is because a typical compiled program includes a lot of code to bootstrap, manage the stack, various library calls etc. So, the code that we are interested in running might be masked out by the uninteresting parts of this boilerplate code, so we only enable stats when we are actually in the function of interest. The cache had changes to allow it being used in a multi-banked setting, but because the baseline design simply uses the caches single banked, these changes are not required for the baseline design.

You are welcome to use our or your components to compose the single core design. Using your own components will count as an extension, so you need to do these changes in `plab5-mcore-ext` directory. You would need to implement these processor changes in order to be able to use your processor, but using your own cache should be fairly out-of-the box.

The composition and the connections of the single core system will look like the Figure 1. The cache request and response ports of the caches connect to the respective i/dcache ports of the processor, while the mem request and response ports connect to outside facing memreq/sp0/1 ports as shown. Note that the data bitwidth from processor to caches is 32 bits, while the data bitwidth from the caches to the test memory is the full cache line, which is 128 bits in this design.

In addition to composing these designs, you are also required to program quicksort as a scalar sorting algorithm. You should implement your sorting algorithm in C in `apps/ubmark/ubmark/ubmark-quicksort.c` in the `quicksort_scalar()` function. Currently, this function only has a template which copies the source array to destination, and your task is to sort the `src` array using quicksort algorithm and write the result to destination. This file also contains verification logic to ensure the correctness of your algorithm (and single core composition). You are welcome to consult textbooks or online resources to learn more about quicksort, but you should cite these resources. Copying code is not allowed; you should write it on your own.

```
            A  |
    memreq0 |  | memresp0
.-----------o--o-----------.
|           |  |           |
|    memreq |  V memresp   |
|        .--o--o--.        |
|        |        |        |
|        | icache |        |
|        |        |        |
|        '--o--o--'        |
|  cachereq A  | cacheresp |
|           |  |           |
|   imemreq |  V imemresp  |
|        .--o--o--.        |
|        |        |        |
|        |  proc  |        |
|        |        |        |
|        '--o--o--'        |
|   dmemreq |  A dmemresp  |
|           |  |           |
|  cachereq V  | cacheresp |
|        .--o--o--.        |
|        |        |        |
|        | dcache |        |
|        |        |        |
|        '--o--o--'        |
|    memreq |  A memresp   |
|           |  |           |
'-----------o--o-----------'
    memreq1 |  | memresp1
            V  |
```

Figure 1: Baseline single core configuration

Instead of debugging your sorting algorithm on your processor, we recommend you do your debugging by compiling natively. Compiling your program natively is very similar to compiling it for your processor. The only difference is the flag you use for the configure flag. Also note that we create a different build directory so that we don't have conflicting binaries for two different ISAs:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/apps
% cd ubmark
% mkdir build-native
% cd build-native/
% ../configure
% make
% ./ubmark-vvadd                        # runs vvadd natively
% ./ubmark-bin-search                   # runs bin-search natively
% ./ubmark-quicksort                    # runs quicksort natively
```

When you run your binaries natively, it will tell you if it passed or failed the verification. If you need further debugging, you can add `printf` statements, or use a debugger such as `gdb`. When you get your app working, make sure you remove any print statements because our architecture does not support it.

In your report, discuss why this is a good baseline design. Describe how you implemented the single core configuration and the quicksort algorithm. Describe any changes you have made to the given design.

3. Alternative Design

The alternative design is to implement the multicore system. The multicore system consists of four processors, four instruction caches that are private to each processor, four data cache banks that are shared among all four cores, and networks to allow multi-banked caches.

We have provided you a memory request/response network in `plab5-mcore-MemNet.v`. Take a look at this module. This module includes adapters to wrap memory messages coming to network messages, a request network, and a response network. In a multi-banked cache design, cache lines are interleaved to different cache banks, so that consecutive cache lines correspond to a different bank. The following is the addressing structure in our multi-banked data caches:

```
+--------------------------+--------------+--------+--------+--------+
|        22b               |     4b       |   2b   |   2b   |   2b   |
|        tag               |   index      |bank idx| offset | subwd  |
+--------------------------+--------------+--------+--------+--------+
```

The memory/network adapters extract this bank index from the memory request address of the memory message to determine the destination of the network, which is the cache bank. In addition, the memory response generated by a cache bank needs to be sent back to the correct processor. To determine who had sent the memory request originally, we tag the memory messages with the requesting source id, i.e., processor id. We use the opaque field to store this information. We overwrite the processor id to the high bits of the opaque field of the memory request message, and pack this memory message in the payload field of a network message. The cache and the memory system is supposed to keep the opaque field of the memory message. Using this, the response network extracts the destination processor id to send the response back.

Another use case of the request/response network is to allow multiple requesters to arbitrate over a single memory port. This is the case in the refill networks for the instruction and data caches, where four instruction caches, and four banks of data caches arbitrate for `memreq0/memresp0` and `memreq1/memresp1` respectively. We parameterize the `MemNet` to set the mode on how it is going to be used. This parameter is called `p_single_bank`, and indicates it should be used as a single-bank setting, i.e. for a refill network. So the refill networks should set this parameter to 1, while the dcache network should set it to 0.

In addition to the network, more changes are necessary in the caches as well. As you might have noticed, the memory addressing fields have changed, and there are bank index bits between the index and the offset. Furthermore, there are fewer tag bits. We have parameterized the cache we have provided (plab3-mem-BlockingCacheBase.v) to tell the number of banks in the system it is going to be used. The parameter is called `p_num_banks`, and this should be set to 1 for instruction caches, and 4 for data caches.

Lastly, the memory request/response network has all of the ports concatenated together for better parameterization. To get the bits for the actual port, we need to use the `` `VC_PORT_PICK_FIELD `` macro.

This macro has two arguments, the first argument is how many bits is each port, the second is the port index. For instance, if you have for the `req_in` port of the dcache network, here is a snippet how you might want to declare the wires and extract the port of interest:

```
// define a local parameter and a short name for the request size
localparam c_proc_req_nbits = ‘VC_MEM_REQ_MSG_NBITS(o,a,d);

localparam prq = c_proc_req_nbits;

wire [prq*p_num_cores-1:0] dcache_net_req_in_msg;
wire [    p_num_cores-1:0] dcache_net_req_in_val;
wire [    p_num_cores-1:0] dcache_net_req_in_val;

plab5_mcore_MemNet
#(
  .p_mem_opaque_nbits   (o),
  .p_mem_addr_nbits     (a),
  .p_mem_data_nbits     (d),

  .p_num_ports          (p_num_cores),

  .p_single_bank        (0)
)
dcache_net
(
  .clk        (clk),
  .reset      (reset),

  .req_in_msg (dcache_net_req_in_msg),
  .req_in_val (dcache_net_req_in_val),
  .req_in_rdy (dcache_net_req_in_rdy),

  // ...
);

// say you want to access the port index 2 for dcache_net_req_in:

assign foo1 = dcache_net_req_in_msg[‘VC_PORT_PICK_FIELD(prq,2)];
assign foo2 = dcache_net_req_in_val[‘VC_PORT_PICK_FIELD(1,  2)];
assign foo3 = dcache_net_req_in_rdy[‘VC_PORT_PICK_FIELD(1,  2)];
```

Software running on different cores need to differentiate themselves from the others. For instance, to parallelize work on an array by dividing and assigning a part of this array to each core, the code can use the core id and the number of cores to assign work. Similarly, to ensure only one of the processors is executing a serial part of the code, software would compare the core id to 0, and only then execute the given code block. To allow this, the processor uses the `mfc0` instruction with the appropriate coprocessor register specifier, which will copy the value of the core id and the number of cores to the specified architectural register. To allow this, each processor needs to know the number of cores in the system and the core id of this processor, using the parameters `p_num_cores` and `p_core_id` respectively.

In addition, to allow synchronization across different cores, the processors and the networks implemented atomic memory operations (AMOs). AMOs enable reading and modifying a memory address *atomically*. Atomicity means that this transaction will be seen to have occurred at the same time with respect to other memory transactions in the same memory address. In other words, there cannot be another memory transaction to the same address while the two transactions, read and a write, of the AMO is going on.

The layout of the components is shown in Figure 2. One thing to mention is the manager interface ports are only connected to `proc0`. This means all of the communication to the outside world (other than the memory) needs to be done through `proc0`.

To ensure that the alternative design works, we use the multi-threaded assembly test suite located in `apps/tests/mt`. These tests are very similar to the single threaded versions, except for testing the correct computation in all of the cores. `proc0` creates work for the other processors, and it waits on all other processors to finish. Instead of directly sending the test outcome like in the case of single threaded assembly tests, each processor writes this outcome to a global array at an index dedicated for this core. Once all of the processors have executed their testing logic, `proc0` checks the test outcomes of each processor. Because `proc0` is the only processor that has its manager ports connected to the test harness, it notifies the manager if any of the cores have failed the test. If all of the cores have passed the test, it sends a pass to the manager. Note that running the single threaded assembly tests would also work in most cases. This would cause all of the processors execute the same code, but only `proc0` will be communicating the pass or fail information back to the manager. The rest of the processors would simply be ignored. So while this does test the single threaded execution on the multicore, to test truly the multi-threaded execution, we need to run the multi-threaded test cases.

In addition to composing the components, you also need to write a parallel sorting algorithm. You are free to pick which sorting algorithm to write. Even though it is parallelizable, quicksort is not the easiest sorting algorithm to parallelize. Instead, we recommend you implement a parallel merge sort algorithm. Merge sort uses a divide-and-conquer approach to initially divide up work, and then build up from there. The merge sort method contains two recursive calls to the merge sort for the low and high halves of the input array. These recursive calls are performed until the arrays are a single element long. Then, for each right and left arrays, which are internally sorted (single element array is sorted), we call a merge function. The merge function simply copies the two sorted input arrays to a combined and sorted destination array. At the exit of each merge sort function, there is a call to the merge function, and the sorted array is returned. This causes these smaller individually sorted arrays to be eventually to be combined into a globally sorted array.

Merge sort is very amenable to parallelization. The input array can be statically divided into the smaller arrays to be sorted. Then each core would simply execute the merge sort algorithm in their quarter of the array. The cores would call `join()` to mark that their quarter of the array is sorted. Finally, these four individually sorted sub-arrays need to be combined to the final sorted array. This reduction step needs to be done single threaded, after the `join()` call. We need to have three final calls to the merge function to combine these four quarter arrays into the one final array.

In your lab report, explain how each component is connected and the additional complexities involved when implementing the alternative design. Will the alternative design perform better on single threaded loads? Will it perform better in multi-threaded loads? How do you thing this architecture perform with fewer or more cores? Do you think it is worth investing in the complexity?

4. Testing strategy

For the testing strategy, unlike the earlier labs, you do not need to come up with additional test cases. Instead, discuss how do we make sure we test the single core and multicore designs effectively. Discuss the different testing techniques you have learned in the labs so far, and how these came together to ensure we have a very sophisticated working design that can execute multi-threaded programs with realistic caches and networks. Discuss the different testing strategies involved to test the single core processor and the multicore, and how do we ensure all four cores, eight caches and three networks are tested both in isolation and together, and how do we know that we are using all four cores for computation?

Also discuss the idea of self-checking assembly tests and how these can be used to test relatively simple instructions such as arithmetic operations, to more complicated ones, such as AMOs. The testing strategy section in your lab report should briefly summarize the approach used in the testing the baseline design before focusing in detail on testing the multicore. Describe how we test the more difficult parts of the multicore design, such as making sure all four cores are being used, and if AMOs actually work.

5. Evaluation

We provide you four single threaded microbenchmarks and you are required to come up with a sorting microbenchmark, with a total of five. All of the single threaded microbenchmarks are in `apps/ubmark/ubmark/` directory with their C source code and the dataset files. Take a look at all of the and familiarize what each one does.

When you compile the benchmarks or assembly tests, the `../convert` and `../convert-cache` commands produce a vmh file for each app that can be loaded to the test memory and run on your processor. In addition, they also produce an assembly dump in `apps/ubmark/build/dump` directory. This file contains the actual assembly instructions the C program compiled into. You can search for the function name (e.g. `vvadd_scalar`) in the dump file to see what the C code got translated to. Compare this compiler generated assembly to the microbenchmarks we have used in Lab 2 which were hand assembled (e.g. `plab2-proc/plab2-proc-ubmark-vvadd.v` file). Which code is more readable? Which code would give better performance? Which code is more optimal in terms of static code size and number of registers used?

In addition to the single threaded benchmarks, we have provided multi-threaded benchmarks in the `apps/ubmark/mtbmark/` directory. We have the same microbenchmarks in the single threaded `ubmark` implemented as multi-threaded. All of these multi-threaded microbenchmarks statically partition the input for the number of cores (threads) that are available in the system. Initially, each core calls `spawn()` which sets up a global array with an entry for each core. Then each cores executes work that is statically assigned. As processors finish executing, they call `join()`, which marks in the global array that the core has finished executing. The processors other than `proc0` which have reached this point then go to an infinite loop. `proc0` busy-waits for all of the remaining processors to call `join()`. Once this is done, this is a safe point to resume single threaded work. Depending on the benchmark, some benchmarks require one final serial reduction step that need to be executed single threaded, using the outputs each core have produced, to produce the final answer. After this optional reduction step, the output is verified, again single threaded.

To see how each of the microbenchmarks perform on both the single-core and multicore systems, you can run the following command:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/build
% make eval-plab5-mcore -j8
```

Because of the complexity of the designs, especially of the multicore, we use the `-j8` flag to parallelize the simulation runs to 8 threads. `amdpool` machines are multicore processors, not too different from what you are implementing, and take advantage of these extra cores! Or alternatively, you can also run each benchmark in isolation:

```
% cd ${HOME}/ece4750/ece4750-lab5-mcore/build
% make plab5-mcore-sim-base
% ./plab5-mcore-sim-base +exe=../apps/ubmark/build/vmh-cache/cache-ubmark-vvadd.vmh +stats
% make plab5-mcore-sim-alt
% ./plab5-mcore-sim-alt +exe=../apps/ubmark/build/vmh-cache/cache-mtbmark-vvadd.vmh +stats
```

When you run a simulation, the simulator will also report whether the verification passed or failed. Make sure all of the verifications pass. It will report the number of instructions (per core) and the number of cycles. Assuming both designs have the same cycle time, we can use the number of cycles as a proxy for the performance. Report these results in your lab report in a table.

You can also experiment with running single threaded benchmarks on the multicore and multi-threaded benchmarks on the single core. These allow you to see the overheads in the software to make a program multi-threaded. Note that when you run single threaded benchmarks on the multicore, some of the benchmarks might not pass the verification. This is because each core will try to execute the same exact code over the same exact data. Depending if the benchmark is written in an idempotent way or not, these additional cores might corrupt each others' data.

In your lab report, discuss how each benchmark performed in each configuration. Did all of the benchmarks perform better on the multicore? Did they get the theoretical speedup of 4X over the single core? If you ran multi-threaded benchmarks on the single core, how much are the overheads to make applications multi-threaded? What are these overheads due to? How do the two versions of sorting perform on these two architectures? Can the parallel sort get a speedup over the serial version of sorting?

6. Extensions

In your single core design, your performance would be considerably worse than Lab 2 you have implemented. The primary reason is that the caches we use have significant hit latency of four cycles. We suggest you improve the hit latency which will directly improve your performance as an extension. You are welcome to use the caches you have used in Lab 3 and improve them further to give a performance boost to the single core processor system.

Other than the caches, you can also experiment with various changes to the processor. You could implement a simple superscalar pipeline, basic multithreading or branch prediction schemes. You can also improve the network by using a crossbar network instead of a ring network.

Note that using your own components from earlier labs also counts as an extension. However, especially for the multicore design, some non-trivial changes were required for both the cache and the processor. But the single core can use relatively unchanged cache and processor implementations. Remember that you should do all of the extension work in a separate extension directory (plab5-mcore-ext).

Acknowledgments

This lab was created by Berkin Ilbeyi, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.

```
                                    A  |
                            memreq0 |  | memresp0
.-----------------------------------o--o---------------------------.
|                                   |  |                           |
|                           req_out |  V resp_in                   |
|     .-----------------------------o--o-----------------------.   |
|     |                  icache refill network                 |   |
|     '-----o--o--------------o--o--------o--o--------o--o-----'   |
|    req_in A  | resp_out     A  |        A  |        A  |         |
|           |  |              |  |        |  |        |  |         |
|    memreq |  V memresp      |  V        |  V        |  V         |
|        .--o--o--.        .--o--o--.  .--o--o--.  .--o--o--.      |
|        |        |        |        |  |        |  |        |      |
|        | icache0|        | icache1|  | icache2|  | icache3|      |
|        |        |        |        |  |        |  |        |      |
|        '--o--o--'        '--o--o--'  '--o--o--'  '--o--o--'      |
|  cachereq A  | cacheresp    A  |        A  |        A  |         |
|           |  |              |  |        |  |        |  |         |
|   imemreq |  V imemresp     |  V        |  V        |  V         |
|        .--o--o--.        .--o--o--.  .--o--o--.  .--o--o--.      |
|        |        |        |        |  |        |  |        |      |
|        |  proc0 |        |  proc1 |  |  proc2 |  |  proc3 |      |
|        |        |        |        |  |        |  |        |      |
|        '--o--o--'        '--o--o--'  '--o--o--'  '--o--o--'      |
|   dmemreq |  A dmemresp     |  A        |  A        |  A         |
|           |  |              |  |        |  |        |  |         |
|    req_in V  | resp_out     V  |        V  |        V  |         |
|     .-----o--o--------------o--o--------o--o--------o--o-----.   |
|     |                   dcache network                       |   |
|     '-----o--o--------------o--o--------o--o--------o--o-----'   |
|   req_out |  A resp_in      |  A        |  A        |  A         |
|           |  |              |  |        |  |        |  |         |
|  cachereq V  | cacheresp    V  |        V  |        V  |         |
|        .--o--o--.        .--o--o--.  .--o--o--.  .--o--o--.      |
|        |        |        |        |  |        |  |        |      |
|        | dcache0|        | dcache1|  | dcache2|  | dcache3|      |
|        |        |        |        |  |        |  |        |      |
|        '--o--o--'        '--o--o--'  '--o--o--'  '--o--o--'      |
|    memreq |  A memresp      |  A        |  A        |  A         |
|           |  |              |  |        |  |        |  |         |
|    req_in V  | resp_out     V  |        V  |        V  |         |
|     .-----o--o--------------o--o--------o--o--------o--o-----.   |
|     |                  dcache refill network                 |   |
|     '-----------------------------o--o-----------------------'   |
|                           req_out |  A resp_in                   |
|                                   |  |                           |
'-----------------------------------o--o---------------------------'
                            memreq1 |  | memresp1
                                    V  |
```

Figure 2: Alternative multicore configuration