



Carnegie Mellon University
School of Computer Science
Computer Science Department

CS740: Computer Architecture

Final Report

FACS: FPGA Accelerated Multiprocessor Cache Simulator

Michael Papamichael [papamix@cs.cmu.edu]

Wei Yu [wy@andrew.cmu.edu]

Yongjun Jeon [yongjunj@andrew.cmu.edu]

Pittsburgh, PA, December 2007

Group Info:

- **Michael Papamichael** [papamix@cs.cmu.edu]
- **Wei Yu** [wy@andrew.cmu.edu]
- **Yongjun Jeon** [yongjunj@andrew.cmu.edu]

Project Web Page:

<http://www.cs.cmu.edu/~mpapamic/projects/facs.html>

Introduction:

Current architectural-level full-system software-based simulators (e.g. Virtutech Simics) are developed to run on single-threaded hosts and are thus limited in throughput, especially when simulating multiprocessor systems. The slowdown becomes even higher when attaching additional modules to the simulator, such as cache models. Simulating a uniprocessor system is typically thousands of times slower than the actual single CPU system, which is still considered an acceptable slowdown by software researchers. However, simulation of a multiprocessor system on a single host is up to a million times slower than the real hardware. This makes execution of large multiprocessor workloads - be it scientific or server workloads - prohibitively slow.

Statistical sampling of computer system simulation [1, 2, 3] can reduce simulation times by roughly a factor of 10000. This is done by employing systematic sampling to measure only a very small portion of the entire workload being simulated. Between measurements, a simplified functional simulation model of the system that also advances the architectural state of the system is used to fast-forward through the workload. Actual measurements are taken using detailed timing-accurate models of the components comprising the simulated system. The use of checkpointing to capture the architectural state of the system can even further accelerate simulation [3].

In addition to statistical sampling, recent research in the field of hybrid simulation has also greatly accelerated multiprocessor system simulation using FPGAs (e.g. Protoflex [4, 5]). This corresponds to arrow 1 in figure 1. Protoflex uses transplant technology to dynamically accelerate only common-case behaviors while relegating infrequent, complex behaviors (e.g., I/O devices) to software simulation. Transplanting avoids having to implement the entire target system in an FPGA. Protoflex is an invaluable

tool for fast-forwarding through large workloads and collecting checkpoints. However attaching additional software-based simulator components, such as cache models, greatly limits simulation speed and throughput of Protoflex.

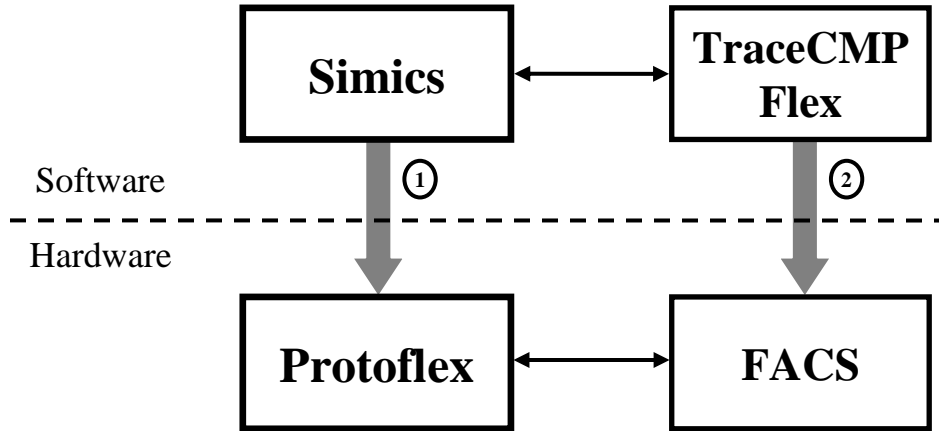


Figure 1: Software vs Hardware simulation

At this point there are two software-based cache models that were developed in the scope of the SimFlex project [1], which both implement a piranha-based cache hierarchy, similar to the one seen in figure 2. “CMPFlex” implements a detailed timing-accurate cache model, while “TraceCMPFlex” is a simplified version of “CMPFlex”, which simulates the piranha-based cache model at a functional level.

FACS Overview

FACS (FPGA-Accelerated Cache Simulator) is a fully parameterizable hardware functional piranha-based [6] multiprocessor cache model that precisely replicates the behavior of the existing software-based TraceCMPFlex cache model. This corresponds to arrow 2 in Figure 1. FACS can receive and process dynamically generated references from Protoflex at full speed, allowing simulation to proceed without any slowdown. For each incoming memory reference FACS updates the cache tags and maintains coherence among the private L1 caches, while also keeping detailed L1 and L2 cache statistics. Our results show that FACS is roughly 200 times faster than the equivalent software functional cache model it simulates. When combined with FACS, ProtoFlex can accelerate a full 16-cpu chip multiprocessor with private 64KB L1 I&D caches and a 16MB shared L2 by a factor of roughly 100x.

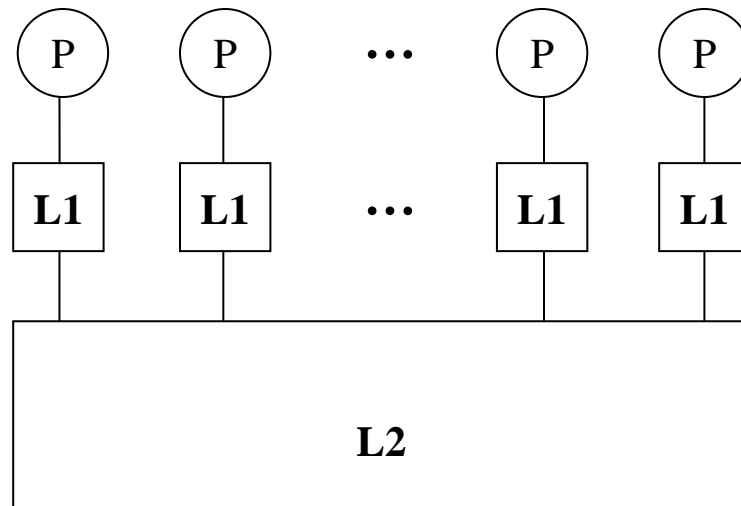


Figure 2: Piranha-based cache hierarchy

FACS was written using the verilog hardware description language and is structured as a 6-stage pipeline which processes streams of memory references (Instr. Fetches, Stores and Loads) received through a FIFO interface. Internally FACS consists of two core modules; one that simulates the set of all private L1 implemented as a 2-stage pipeline and one that simulates the single shared L2 cache implemented as a 4-stage pipeline. FACS's pipeline-based design greatly reduces hardware complexity and increases memory reference throughput.

Since FACS implements a purely functional cache model it only stores and updates the tags and the status bits for each cache-line. Not storing the data for each cache-line leads to better memory utilization on the actual hardware (e.g. the FPGA) allowing for the implementation of larger cache designs and a more scalable design in general. As an example modeling 16 128KB L1 caches, along with a 16MB L2 cache requires less than 500KB of memory, which can easily fit on on-chip FPGA RAM.

Methodology

Hardware

FACS required a significant amount of hardware design, verification and implementation, as well as software development. The first task was to design the necessary hardware and describe it using the

verilog hardware description language. Next we had to verify the correct behavior of the created verilog modules using the verilog simulator Modelsim 6.1e. This was done in two steps. Firstly, we thoroughly observed and checked the behavior of our module by feeding small sets of memory references, which were manually created and tested various corner cases. Secondly, we fed our module very large traces of memory references that were generated using the software-based simulator Flexus and made sure that the results obtained by our hardware cache model match with the software-generated results. More details on the verification of our cache model can be found in Appendix A.

After finishing with the verification of our verilog module using simulation the next step was to try it on the real hardware. For synthesis, placement and routing we used the Xilinx ISE 9.2i software. Our first target was the Xilinx University Program (XUP) development board [7] which is based on a relatively small Xilinx FPGA (VirtexIIPro xc2vp30). For our initial test we synthesized a stripped-down version of our cache model, which had all 32 – 16 for Instructions and 16 for Data - L1 caches but no L2 cache. A small set of static references was fed to the design and our statistics registers were observed using the Xilinx Chipscope 9.2i software. Chipscope is a tool that allows real-time monitoring of FPGA signals.

After verifying our stripped-down cache model using the XUP board, we started working on the more sophisticated BEE2 development board [8] that hosts 5 larger Xilinx FPGAs (VirtexIIPro xc2vp70), that could fit our whole design. In order to feed our design with larger memory reference traces we utilized the PowerPC processor which is embedded in each FPGA using the Xilinx EDK 9.2i software. In order for the PowerPC to be able to push references to our design we presented our cache model to the PowerPC as a memory-mapped peripheral. This required developing an additional PLB-IPIF [9] interface for our cache model that allows the PowerPC to push references, read and write the statistics memories, and read out the contents of the cache.

Each memory reference takes up 40 bits, but the IPIF interface is only 32 bits wide. To avoid having the PowerPC make two stores for pushing each reference to our cache model we made the following optimization. We mapped a larger portion of the physical address space to our cache model and encoded the additional 8 bits as part of the address. This way the effective data width for communicating with our cache model was increased to 48 bits. In practice this optimization almost doubled the speed at which the PowerPC is able to feed references.

In our first tests using the BEE2 board we used FPGA on-chip memory (BRAM) to store the traces. This limited the maximum traces we could try to approximately 10000 memory references. In addition this also limited us from placing larger cache models on the FPGA, since many FPGA resources

(BRAMs) had to be occupied by the traces. To solve this problem we utilized the onboard DDR2 DRAM memory found on the BEE2 board. This increased the maximum number of references to approximately 4 million per run. As a final step for feeding even larger traces to our cache model we developed software that allows the PowerPC to read traces that reside on compact flash cards. The maximum number of traces is now only limited by the size of the individual compact flash card.

Software

The main software tool we used for FACS was Flexus. Flexus is a collection of modules that attach to Virtutech Simics' Micro-Architecture Interface (MAI) for full-system, timing-accurate simulation of multiprocessor systems running unmodified programs. It also has built-in statistics management and supports checkpointing. Flexus is written in C++ with the extensive use of the Boost library. At the top level, Flexus is invoked using a startup script with a user configuration, a specific simulator and a workload.

A simulator is defined by a wiring.cpp file, which actually reads more like a verilog source file, that instantiates and wires together various components such as caches and branch predictors, whose default parameters, such as size and associativity can be overridden at runtime to match those of the HW model. The individual modules are hooked together during compilation.

A Flexus component is an individual module that is instantiated by a simulator. The most important Flexus component for FACS was TraceCMPFlex, which is a functional Piranha-based multiprocessor cache simulator. Below are more details about a few Flexus components that we extensively used in the FACS project.

- **DecoupledFeeder**

In Flexus, the DecoupledFeeder module connects the processors and the L1 caches. It receives a serial stream of memory references from the processors and pushes them into the appropriate caches. Most of our modifications were in this module, and the modified version provides the timing information and supports TRACE-OUT and TRACE-IN modes in which the module outputs and takes as input serialized memory references in a file. The exact format of the memory references file is described in Appendix A.

- **FastCache**

The FastCache module corresponds to each one of the simulated L1 Instruction and Data caches. In terms of the statistics counters, we were able to achieve exact behavioral replication of L1 caches without any modifications to the FastCache module itself.

- **FastCMPCache**

The FastCMPCache is instantiated as the Piranha-based functional L2 cache in the Flexus simulator. It is worth noting that this module, by default, classifies some of the statistics a bit differently from our HW module, and a modification was necessary to get the statistics to perfectly match up.

- **Stat-manager**

The stat-manager is a built-in statistics management tool for analyzing the results of a workload simulation. By default, it reports only the aggregate statistics for all of the L1 caches, so it had to be modified to preserve and report the statistics from all individual L1 caches.

Implementation

FACS Architecture

The architecture of FACS at the module-level is depicted in the block diagram of figure 3. As seen in the previous section at the highest level FACS consists of two core modules; one that simulates all of the L1 caches and one that simulates the L2 cache. The “L1 Caches” module contains two identical instances of the same submodule, one for implementing the Instruction and one for implementing the Data caches. Each such submodule (Instruction or Data) instantiates a statistics module and 16 “L1cache” modules. Finally, the L1cache module represents the private cache of each CPU in the system and contains the LRU logic along with two submodules that simulate each one of the two ways that belong to the L1 cache of each processor. The L2 cache module is quite simpler; it instantiates the 8 2MB ways, implements the LRU logic and also keeps statistics.

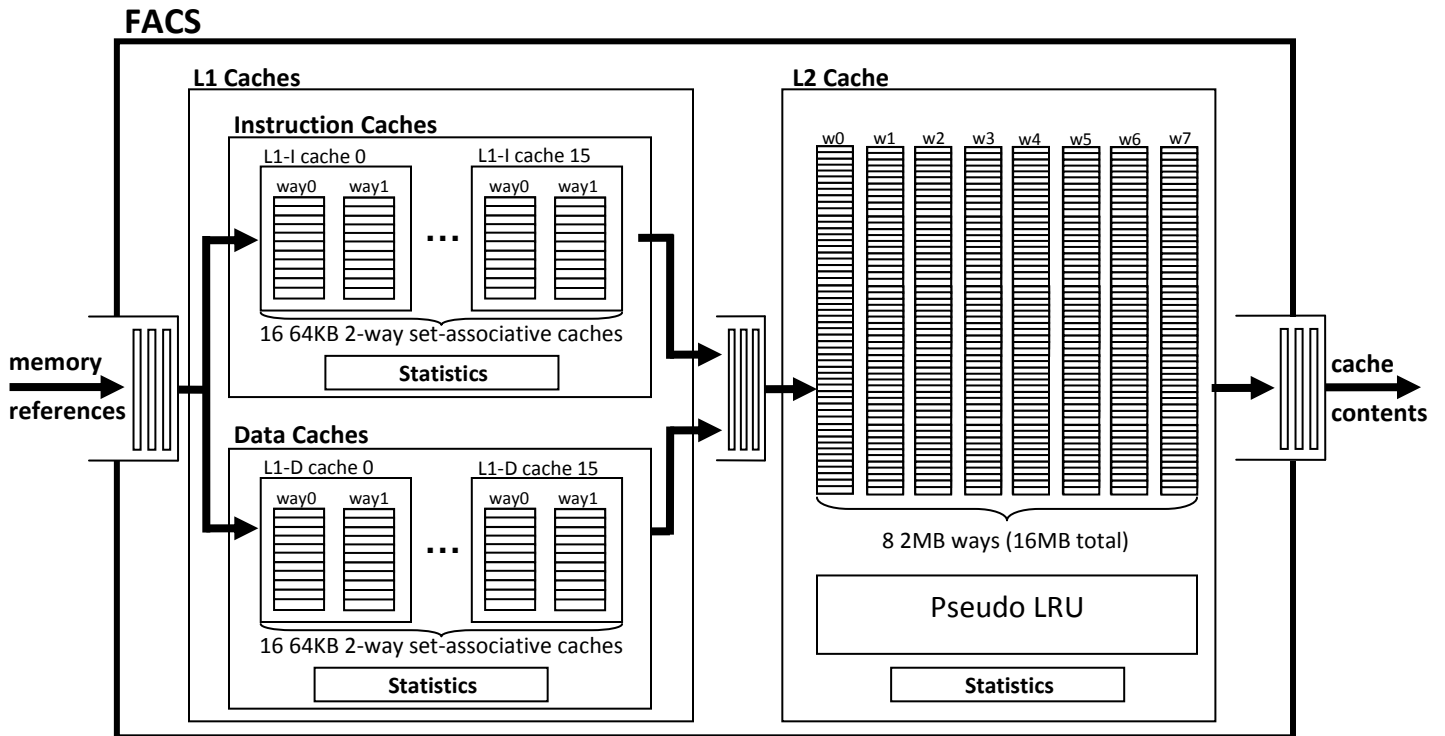


Figure 3: Architecture of FACS

The actual implementation contains numerous other basic modules, such as FIFO queues, that act as building blocks for creating the higher-level modules mentioned above. FACS uses FIFO queues for receiving references from the “outside world”, forwarding references from the “L1 caches” to the “L2 cache” module and for scanning out the actual L1 and L2 cache contents. Other examples of basic modules include fully parameterizable decoders, multiplexors and dual-port memories.

L1 Caches

The default L1 cache configuration in FACS consists of 32 2-way set-associative 64KB L1 caches; 16 dedicated to data and 16 for instructions. The default size of each cache block is 64 bytes. In an actual implementation of a piranha-based cache a copy of the L1 cache tags is stored in the L2 cache to be used as a directory for maintaining coherence. When a CPU misses in its local L1 cache the memory reference travels to the L2 cache and consults the directory to find out if a remote L1 cache has a copy and take appropriate coherence actions. However, given the fact that ProtoFlex generates memory references serially, FACS is able to follow a simpler approach.

For each received memory reference all 64 cache ways are simultaneously accessed. Regardless of the memory reference type, both Instruction and Data caches are accessed to maintain Instruction and Data cache coherence. If the local cache hits then only the LRU needs to be updated. If the local L1 cache experiences a miss, the remote L1 caches are updated to maintain coherence and the memory reference is forwarded to the L2. In case of a read miss the remote caches downgrade their copy to the shared state. In case of a write miss the remote caches must invalidate their copy. Contrarily, in a real system the remote L1 caches would only be updated after the reference reached the directory residing in L2.

Apart from updating the tag and status bits for the respective referenced L1 cache block the L1 cache module also updates the LRU memory, and updates the statistics counters. The following 5 statistics counters are kept for each L1 cache:

- Number of Read (load) Hits
- Number of Write (store) Hits
- Number of Read (load) Misses
- Number of Write (store) Misses
- Number of Write (store) Misses that lead to an upgrade (local cache had read-only copy)

As stated earlier the L1 cache module is implemented as a 2-stage pipeline. In the first pipeline stage all of the L1 caches are read and in the second pipeline stage the updated values are written back. It is important to note that once a memory reference goes through this 2-stage pipeline it will never “bother” the L1 cache module again.

L2 Cache

In piranha-based cache schemes the L2 caches act as victim caches. This means that blocks are inserted in the L2 cache only when they are evicted from some L1 cache. In the specific cache model (TraceCMPFlex), blocks are also inserted into the L2 cache to facilitate block transfers from one L1 to another L1 cache. For instance, if a CPU attempts to write a cache block that also resides in a remote L1 cache, the cache block will also be inserted into the L2 cache. As far as LRU is concerned FACS precisely replicates the pseudo-LRU algorithm used in the software implementation of the cache model (TraceCMPFlex).

The default L2 cache configuration in FACS consists of an 8-way set-associative 16MB shared cache, which is common for both data and instructions. The default size of each cache block is 64 bytes (equal to the L1 cache block). In an actual piranha-based L2 cache implementation, the L2 cache would also keep copies of all L1 cache tags. However in FACS L1 caches are already fully updated and coherent

when the memory references reach the L2 cache. Thus the L2 cache does not need to keep any additional information; it only stores the tags and status bits for its own cache blocks.

Each memory reference that misses in its local L1 cache is forwarded to the L2 cache module, which, as mentioned previously, is implemented as a 4-stage pipeline. In the first 2 pipeline stages FACS checks if the initially requested block resides in the L2 cache. If this block was found in another L1 cache, then it has to also be inserted into the L2 cache to simulate L1 to L1 block transfers. During the remaining 2 pipeline stages the block that was evicted from L1 is inserted into the L2 cache. As each memory reference is processed by the L2 the LRU memory and statistics counters are also updated.

The following statistics counters are kept for the L2 cache:

- Number of Read (load) Hits
- Number of Write (store) Hits
- Number of Read (load) Misses
 - Served from main memory
 - Served from another L1 cache
- Number of Write (store) Misses
 - Served from main memory
 - Served from another L1 cache
- L2_Victim_Hits
- L2_Victim_Misses

Parameters

FACS was written as a fully parameterized set of modules, allowing for effortless experimentation with different settings. Parameters include:

- Number of Address Bits (default value: 32)
- Number of CPUs/L1 Private Caches (default value: 16)
- L1 Block Size (default value: 64 bytes)
- L1 Way Size (default value: 32KB)
- L2 Block Size (default value: 64 bytes)
- L2 Associativity/Number of Ways (default value: 8 ways)
- L2 Way Size (default value: 2MB)
- Statistics Granularity (default value: 32 bits)

Software Development

FACS also required a significant amount of software development, which mainly concerned Flexus, as mentioned in the software methodology section above. In addition, a couple of additional tools were developed to help analyze the cache statistics results from FACS and Flexus:

Converter

Converter is a simple C++ program to translate the dumped memory references from Flexus into the binary format recognizable by FACS. The devising of a binary format involved a fair amount of hacking to overcome the architectural characteristics of the PowerPC on the BEE2 board, as mentioned in the hardware methodology section above.

Comparator

Comparator is a simple Python script to automate the comparison of the L2 and the 16 L1 I- and D-cache statistics from FACS and Flexus. It reports the total references from each model and displays any discrepancies. A thorough analyses of the L1 and L2 statistics is presented in Appendix A.

Results

In order to evaluate FACS we compared the time it takes to process references using the software TraceCMPFlex cache model against the time it takes to process the references using FACS. Figures 4 and 5 plot the required time to process millions of references and Figure 6 shows the equivalent speedup values. For the Flexus TraceCMPFlex software-based cache simulator we plot two curves, which correspond to the execution on two different machines with the following specifications.

- An Intel Xeon @ 2.8GHz with 512KB L2 cache and 3GB RAM (2 cores in total; machine **tamdhu** in the ECE Scotch cluster)
- A dual Intel Xeon 5130 running at 2GHz with 4MB shared L2 cache and 8GB RAM (4 cores in total; machine **brackla** in the ECE Scotch cluster)

For FACS, our hardware cache model, we also plot two separate curves. The first curve corresponds to the common case, where the average L1 cache hit rate is above 75%. This is typical for most programs and was true for all of the workloads that we tried. The second curve corresponds to the worst case, where

all references miss in the L1 cache (0% L1 cache hit rate). In either case FACS is roughly 200 times faster than the software cache simulator, regardless of which machine is it executed on.

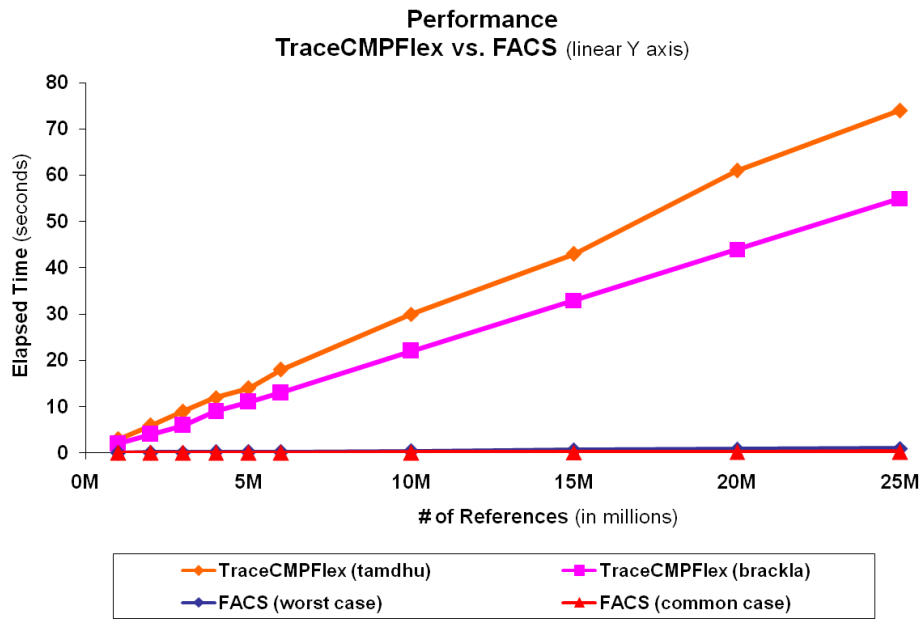


Figure 4: Performance of FACS vs. TraceCMPFlex

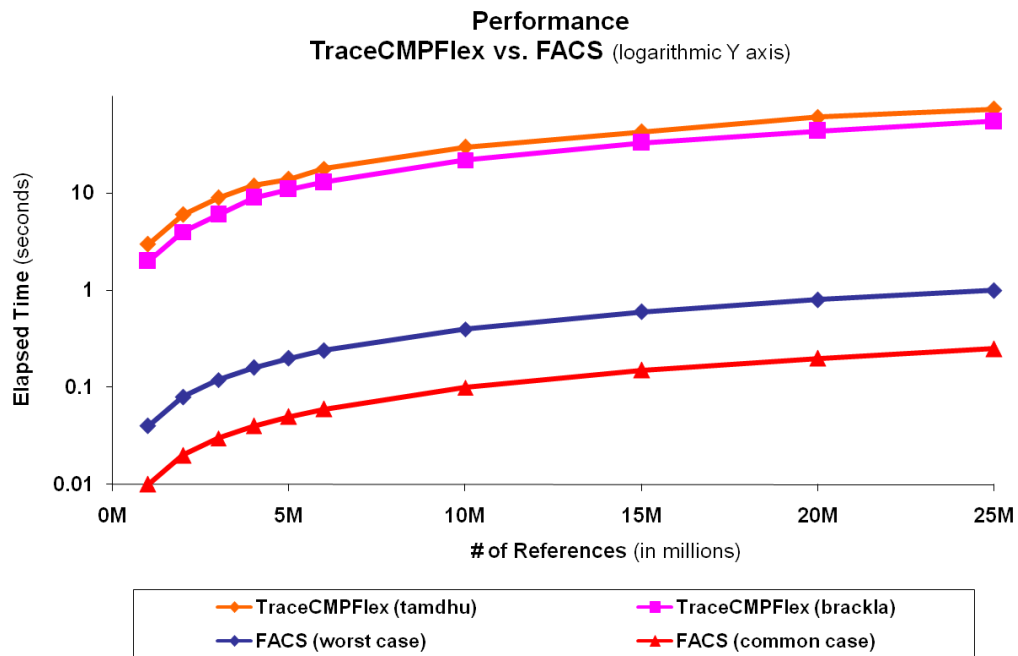


Figure 5: Performance of FACS vs. TraceCMPFlex

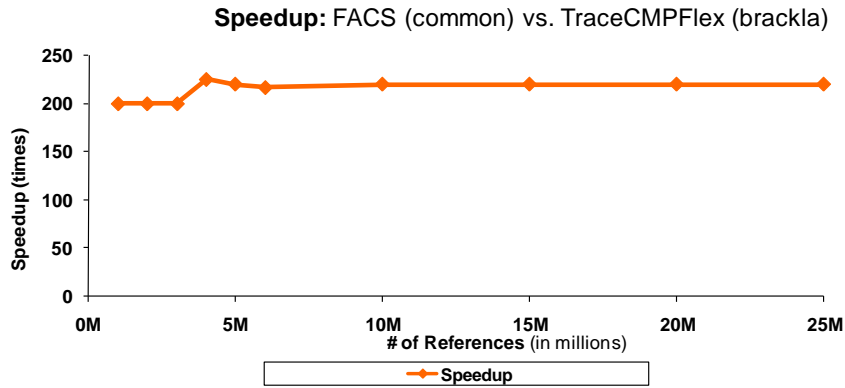


Figure 6: Performance of FACS vs. TraceCMPFlex

Figure 7 plots the time required for FACS to process a varying number of references for both the common case and the worst case. Clocked at 100MHz, FACS can process L1 requests through its 2-stage pipeline at 100 million references per second. However for the L2 cache each reference requires 4 cycles to be processed. As an effect the actual processing rate of the FACS L2 cache is 25 million references per second. However in almost all cases this does not slow down the aggregate throughput of FACS, since the L1 caches act as “reference filter” for the L2 cache. In other words only L1 cache misses will be forwarded to the L2 cache. If the average L1 hit rate is above 75% then FACS can achieve its peak throughput of 100 million references per second. In the very rare case, where the L1 average miss rate is above 25%, FACS throughput will decrease. The worst case scenario, where every memory reference misses in the L1 caches and has to travel to the L2 cache, limits the throughput of FACS to 25 million references per second, which is still enormously higher than what the software can achieve.

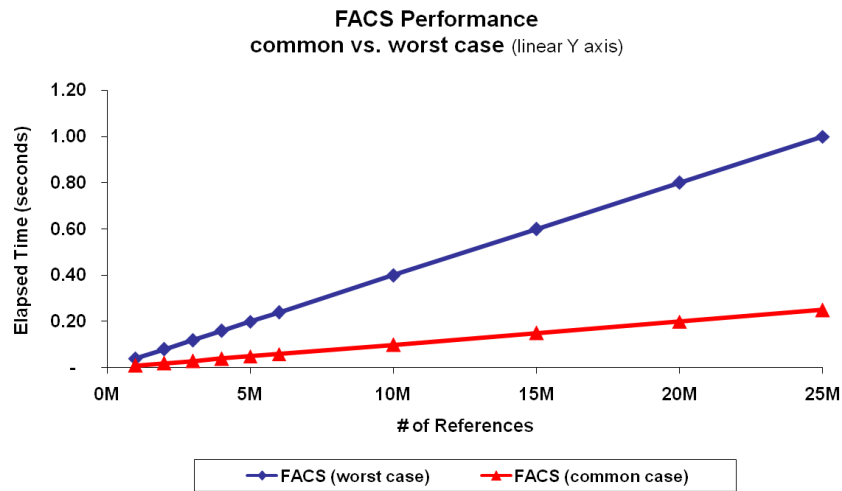


Figure 7: Performance of FACS: Common vs. Worst Case

Conclusions & Future Work

In this report we presented FACS, a fully parameterizable FPGA-based hardware piranha-based multiprocessor cache model that precisely replicates the behavior of the existing software-based TraceCMPFlex cache model. Our performance results show that FACS outperforms the software cache model by 2 to 3 orders of magnitude, achieving an aggregate throughput of 100 million references per second and a speedup of roughly 200x over the software model. FACS is a proof of concept project that shows the potential of FPGA-accelerated simulation. FACS can be easily integrated with the hybrid FPGA-accelerated simulator ProtoFlex to achieve speedups of about 100x when compared to Simics and the TraceCMPFlex software cache model. We are very satisfied with the results of our project and are proud to say that we surpassed our initial 125% goal and also partially reached our 150% goal.

FACS opens up many opportunities for future work. As a first step one could add finer grain statistics counters to get a better understanding of the detailed cache behavior. Another interesting direction would be to build a distributed hardware cache model that utilizes more than one FPGA. This would allow for a more scalable solution that allows the simulation of larger multiprocessor cache systems. Even further, it would be interesting to experiment with other multiprocessor cache models and see how easily they could be ported to hardware and what speedup they can achieve.

Acknowledgements & Distribution of Credit

We would like to thank Eric Chung for his support on the BEE2 development board and Nikos Hardavellas for helping us out with the TraceCMPFlex software cache model. We would also like to thank James Hoe for providing us with previously developed pieces of verilog code that greatly reduced the required implementation time.

The distribution of credit for each part of FACS is given below.

- **Hardware Simulation and Verification**
 - 50% : Michael Papamichael
 - 50% : Wei Yu
- **Hardware Implementation**
 - 60% : Michael Papamichael
 - 40% : Wei Yu

- **Software Development**

- 80% : Yongjun Jeon
- 10% : Wei Yu
- 10% : Michael Papamichael

Regardless of the numbers above we would like to note that this project was a result of well coordinated team effort and that the contribution of all group members was essential for the completion of the project.

Bibliography

[1] SIMFLEX: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture

Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisich, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzky.

In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31, No. 4, pp. 31-35, March 2004

[2] SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling

Roland E. Wunderlich, Thomas F. Wenisich, Babak Falsafi and James C. Hoe.

In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003

[3] TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes

Thomas F. Wenisich, Roland E. Wunderlich, Babak Falsafi and James C. Hoe.

Poster in the *International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS 2005)*, June 2005

[4] ProtoFlex: FPGA-accelerated Hybrid Functional Simulation

Eric S. Chung, Eriko Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai.

CALCM Technical Report 2007-2, February 2007

[5] ProtoFlex: Co-Simulation for Component-wise FPGA Emulator Development

Eric S. Chung, James C. Hoe, and Babak Falsafi

In the *2nd Workshop on Architecture Research using FPGA Platforms (WARFP 2006)*, February 2006

[7] Xilinx University Program (XUP) Board: <http://www.xilinx.com/univ/xupv2p.html>

[6] BEE2 Board: <http://bee2.eecs.berkeley.edu/>

[8] Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing

Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzky, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, Ben Verghese.

Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA 00), IEEE CS Press, 2000, pp. 282-293.

[9] PLB IPIF: http://www.xilinx.com/products/ipcenter/plb_ipif.htm

Appendix A

Verifying our cache model

A large portion of our time and effort in this project went towards verifying the functional correctness of FACS. We used Modelsim SE 6.1e to simulate our verilog hardware model and compared the hardware statistics results with the statistics obtained from Flexus and the TraceCMPFlex software cache model.

Reference Generator

We used simpler manually generated traces, as well as larger traces generated by Flexus using the TRACE-OUT mode of the modified DecoupledFeeder component. Each line in a trace file represents a memory reference, represented in the following format:

`<CPUID> <ADDRESS> <RorW> <IorD>`

where RorW and IorD are 1-bit values indicating whether a particular reference is a memory read or write (RorW=1 or RorW=0) and whether it should be sent to the instruction or the data cache (IorD=1 or IorD=0). CPUID is a 4-bit quantity that identifies the processor that made the request (0-15) and ADDRESS is a 32-bit quantity used to store the reference address.

There are four possible different types of memory references:

1. Data Read (Load): For instance “**6 136d369c 1 0**” indicates that CPU 6 issued a data load from memory address 0x136D369C.
2. Data Write (Store): For instance “**4 1324c345 0 0**” indicates that CPU4 issued a data store to memory address 0x1324C345.
3. Instruction Read (Fetch): For instance “**9 a0f9790 1 1**” means CPU9 issued a fetch from memory address 0xA0F9790.
4. Instruction Write: While instruction writes are theoretically possible they do not appear in actual workloads. However they are supported by FACS.

The generated traces were fed to both FACS and Flexus using the TRACE-IN mode of the DecoupledFeeder component. The statistics from both models were then collected and compared against each other using the comparator tool described in the software implementation section.

Details about the Statistics

TraceCMPFlex collects very detailed statistics for its cache model. After simulating the execution of a workload, the statistics are dumped to the stats_db.out.gz file. Out of the many statistics that are kept by TraceCMPFlex the following are of interest to us:

L1cache statistics	L2 cache statistics
<i>NN</i> -L1[<i>I,D</i>]-Hits:Read	sys-L2-Hits:Fetch
<i>NN</i> -L1[<i>I,D</i>]-Hits:Write	sys-L2-Hits:Read
<i>NN</i> -L1[<i>I,D</i>]-Misses:Read	sys-L2-Hits:Write
<i>NN</i> -L1[<i>I,D</i>]-Misses:Write	sys-L2-Hits:Custom:From:kUpgradeMiss:WriteReq
<i>NN</i> -L1[<i>I,D</i>]-Misses:Upgrade	sys-L2-Misses:Fetch
where <i>NN</i> is the CUID	sys-L2-Misses:Read
	sys-L2-Misses:Write

The L1 cache statistics obtained from Flexus and TraceCMPFlex match directly to those from FACS. For the L2 cache statistics, some additional work needs to be done. FACS maintains four counters for L2 cache statistics:

1. L2_MissesfromL1:Read
2. L2_MissesfromL1:Write
3. L2_MissesfromMem:Read
4. L2_MissesfromMem:Write

L2_MissesfromMem:[Read, Write] corresponds to the case where an L1 cache has a local miss, and there are no other L1 caches that contain the requested block. In this case, L1 needs to bring in the block from the main memory, and the block gets stored directly into the requesting L1 cache and not the L2 cache.

L2_MissesfromL1:[Read, Write] corresponds to the case where an L1 faces a local miss, and the requested block resides in the L1 cache of another processor. In this case, the requesting L1 cache brings in the block from the other L1 cache, and the requested block is also inserted into the L2 cache.

The following tables summarize the relationship between the TraceCMPFlex and FACS statistics:

For the L1 cache statistics there is a direct mapping:

TraceCMPFlex	FACS
NN-L1[I,D]-Hits:Read	= NN-L1[I,D]-Hits:Read
NN-L1[I,D]-Hits:Write	= NN-L1[I,D]-Hits:Write
NN-L1[I,D]-Misses:Read	= NN-L1[I,D]-Misses:Read
NN-L1[I,D]-Misses:Write	= NN-L1[I,D]-Misses:Write
NN-L1[I,D]-Misses:Upgrade	= NN-L1[I,D]-Misses:Upgrade
where NN is the CPUID	

For the L2 cache statistics the following formulas have to be used:

TraceCMPFlex	FACS
sys-L2-Hits:Fetch + L2-Hits:Read	= L2_Hits:Read + L2_MissesfromL1:Read
sys-L2-Hits:Write + sys-L2-Hits:Custom:From:kUpgradeMiss:WriteReq	= L2_Hits:Write + L2_MissesfromL1:Write
sys-L2-Misses:Fetch + L2-Misses:Read	= L2_MissesfromMem:Read
sys-L2-Misses:Write	= L2_MissesfromMem:Write

As mentioned earlier, static traces are generated using the TRACE-OUT mode of the DecoupledFeeder component. We fed the static traces into TraceCMPFlex and FACS and analyzed the cache statistics from both. We also present results from large trace files, for purpose of thorough testing.

Simple memory reference traces

The following table shows the contents of a simple trace file.

Reference Number	CPU ID	Address	RorW	IorD
1	1	103C0	1	0
2	1	103C2	0	0
3	1	103C4	1	0
4	2	103C6	1	0
5	2	103C8	0	0
6	2	183C8	1	0
7	2	183C8	1	0
8	2	183C8	1	0
9	2	183C8	1	0

Here is what happens at each instruction.

1. CPU 1 reads (loads) from 0x103C0 (TAG=0x2, INDEX=0xf, OFFSET=0x0)
Expected result is a L1 read miss.
2. CPU 1 writes (stores) to 0x103C2 (TAG=0x2, INDEX=0xf, OFFSET=0x2)
Expected result is a write miss that causes an upgrade.
3. CPU 1 reads (loads) from 0x103C4 (TAG=0x2, INDEX=0xf, OFFSET=0x4)
Expected result is a read hit.
4. CPU 2 reads (loads) from 0x103C6 (TAG=0x2, INDEX=0xf, OFFSET=0x6)
Expected result is a read miss that will also downgrade the CPU1 cache block.
5. CPU2 writes (stores) to 0x103C8 (TAG=0x2, INDEX=0xf, OFFSET=0x8)
Expected result is a write miss that causes an upgrade and also invalidates CPU1 cache block.

6. CPU 2 Reads (loads) from 0x183C8 (TAG=0x3, INDEX=0xf, OFFSET=0x8)
Expected result is a read miss that fills the second way of the cache set at index 0xF.
7. CPU 2 Reads (loads) from 0x103C8 (TAG=0x2, INDEX=0xf, OFFSET=0x8)
Expected result is a read hit.
8. CPU 2 Reads (loads) from 0x283C8 (TAG=0x5, INDEX=0xf, OFFSET=0x8)
Expected result is a read miss that replaces the second way of cache (replacing 0x183C8).
9. CPU 2 Reads (loads) from 0x183C8 (TAG=0x3, INDEX=0xf, OFFSET=0x8)
Expected result is a read miss.

Figure A-1 shows the Modelsim waveform window, where we can observe the following events:

1. CPU1 gets a read miss
2. CPU1 gets a write miss upgrade
3. CPU1 gets a read hit
4. CPU2 gets a read miss
5. CPU2 gets a write miss upgrade
6. CPU2 gets a read miss
7. CPU2 gets a read hit
8. CPU2 gets a read miss
9. CPU2 gets a read miss

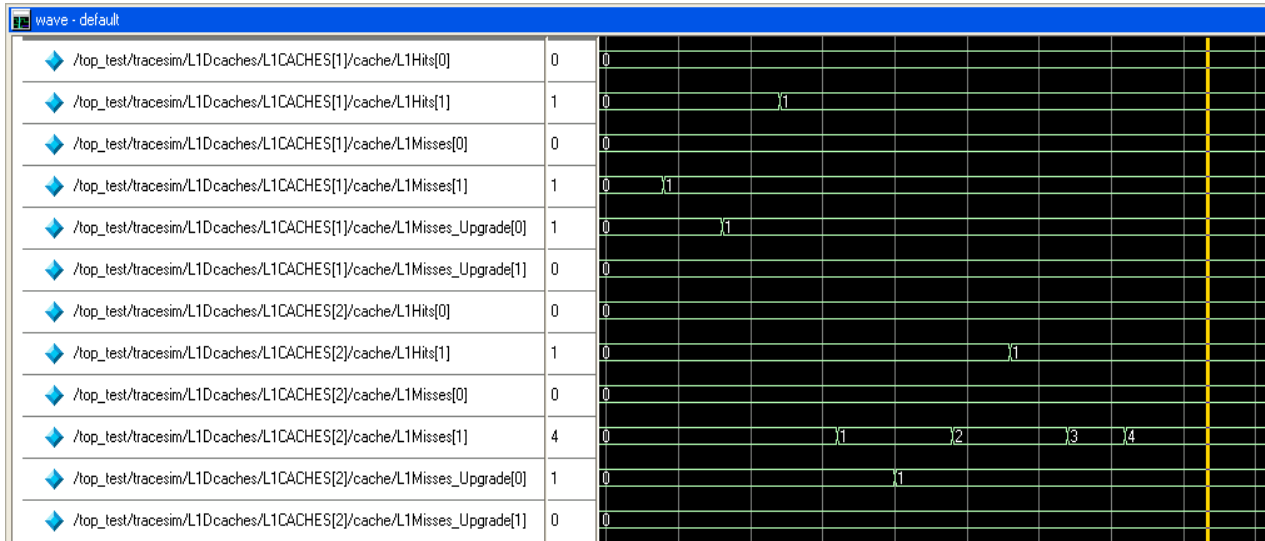


Figure A-1: Modelsim simulation wave window for pathological case

Figure A-2 is a snapshot of the Flexus and TraceCMPFlex console output:

Nodes-L1d-Hits:Atomic	0
Nodes-L1d-Hits:Evict	0
Nodes-L1d-Hits:Fetch	0
Nodes-L1d-Hits:Prefetch	0
Nodes-L1d-Hits:Read	2
Nodes-L1d-Hits:Upgrade	0
Nodes-L1d-Hits:Write	0
Nodes-L1d-Misses:Fetch	0
Nodes-L1d-Misses:Prefetch	0
Nodes-L1d-Misses:Read	5
Nodes-L1d-Misses:Upgrade	2
Nodes-L1d-Misses:Write	0

Figure A-2: Flexus statistics

Large memory reference traces

We also tested both models using large trace files generated from Flexus by running multiprocessor server and scientific workloads. Here we show statistics from both the software TraceCMPFlex cache model and the FACS hardware cache model running a 16-cpu version of the apache webserver with 40 clients.

L1 statistics

Table A-1 shows the sum of individual counters for 16 processors. The corresponding counters from TraceCMPFlex and FACS match perfectly.

	<i>1 million mem refs</i>		<i>2 million mem refs</i>		<i>3 million mem refs</i>		<i>4 million mem refs</i>	
	SW	HW	SW	HW	SW	HW	SW	HW
<i>L1D-Hits:Read</i>	<i>145740</i>	<i>145740</i>	<i>288006</i>	<i>288006</i>	<i>439323</i>	<i>439323</i>	<i>594804</i>	<i>594804</i>
<i>L1D-Hits:Write</i>	<i>62305</i>	<i>62305</i>	<i>124439</i>	<i>124439</i>	<i>183886</i>	<i>183886</i>	<i>242926</i>	<i>242926</i>
<i>L1D-Misses:Read</i>	<i>8646</i>	<i>8646</i>	<i>16309</i>	<i>16309</i>	<i>24408</i>	<i>24408</i>	<i>31105</i>	<i>31105</i>
<i>L1D-Misses:Write</i>	<i>2442</i>	<i>2442</i>	<i>4679</i>	<i>4679</i>	<i>6436</i>	<i>6436</i>	<i>7925</i>	<i>7925</i>
<i>L1D-Misses:Upgrade</i>	<i>2649</i>	<i>2649</i>	<i>5267</i>	<i>5267</i>	<i>7900</i>	<i>7900</i>	<i>10151</i>	<i>10151</i>
<i>L1I-Hits:Read</i>	<i>762261</i>	<i>762261</i>	<i>1532763</i>	<i>1532763</i>	<i>2296802</i>	<i>2296802</i>	<i>3060975</i>	<i>3060975</i>
<i>L1I-Hits:Write</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
<i>L1I-Misses:Read</i>	<i>15957</i>	<i>15957</i>	<i>28537</i>	<i>28537</i>	<i>41245</i>	<i>41245</i>	<i>52114</i>	<i>52114</i>
<i>L1I-Misses:Write</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
<i>L1I-Misses:Upgrade</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>

Table A-1: L1 cache statistics from TraceCMPFlex and FACS

L2 statistics

Table A-2 shows the L2 cache statistics from TraceCMPFlex and FACS. Once again after applying the above mentioned formulas the software and hardware results match perfectly.

1 million				2 million			
SW		HW		SW		HW	
L2Hits:Fetch	10611	L2Hits:Read	9975	L2Hits:Fetch	21392	L2Hits:Read	23588
L2Hits:Read	2896	L2MissesfromL1:Read	3532	L2Hits:Read	6705	L2MissesfromL1:Read	4509
L2Hits:Fetch+L2Hits:Read	13507	L2Hits:Read+L2MissesfromL1:Read	13507	L2Hits:Fetch+L2Hits:Read	28097	L2Hits:Read+L2MissesfromL1:Read	28097
L2Hits:Write	217	L2Hits:Write	179	L2Hits:Write	548	L2Hits:Write	482
L2Hits:Custom	13	L2MfL1:Write	51	L2Hits:Custom	50	L2MfL1:Write	116
L2Hits:Read+L2Hits:Custom	230	L2Hits:Write+L2MfL1:Write	230	L2Hits:Read+L2Hits:Custom	598	L2Hits:Write+L2MfL1:Write	598
L2Misses:Fecth	5346	L2MissesfromMem:Read	11096	L2Misses:Fecth	7145	L2MissesfromMem:Read	16749
L2Misses:Read	5750			L2Misses:Read	9604		
L2Misses:Ft+Rd	11096			L2Misses:Ft+Rd	16749		
L2Misses:Write	2212	L2MfMem:Write	2212	L2Misses:Write	4081	L2MfMem:Write	4081
3 million				4 million			
SW		HW		SW		HW	
L2Hits:Fetch	33711	L2Hits:Read	39555	L2Hits:Fetch	44111	L2Hits:Read	53434
L2Hits:Read	11319	L2MissesfromL1:Read	5475	L2Hits:Read	15257	L2MissesfromL1:Read	5934
L2Hits:Fetch+L2Hits:Read	45030	L2Hits:Read+L2MissesfromL1:Read	45030	L2Hits:Fetch+L2Hits:Read	59367	L2Hits:Read+L2MissesfromL1:Read	59368
L2Hits:Write	1030	L2Hits:Write	907	L2Hits:Write	1558	L2Hits:Write	1433
L2Hits:Custom	82	L2MfL1:Write	205	L2Hits:Custom	127	L2MfL1:Write	252
L2Hits:Read+L2Hits:Custom	1112	L2Hits:Write+L2MfL1:Write	1112	L2Hits:Read+L2Hits:Custom	1685	L2Hits:Write+L2MfL1:Write	1685
L2Misses:Fecth	7534	L2MissesfromMem:Read	20623	L2Misses:Fecth	8003	L2MissesfromMem:Read	23851
L2Misses:Read	13089			L2Misses:Read	15848		
L2Misses:Ft+Rd	20623			L2Misses:Ft+Rd	23851		
L2Misses:Write	5324	L2MfMem:Write	5324	L2Misses:Write	6240	L2MfMem:Write	6240

Table A-2: L2 cache statistics from Flexus and FACS.