Appendix A | **Project management**

In this appendix we present the effective schedule and efforts of the Master's Thesis.

This Master's Thesis has been developed since January until September 2012.

Figure A.1 shows the detailed schedule (calendar) with the different tasks involved.
This Master's Thesis might be divided in the following tasks:

- Literature review. In order to implement a realistic fetch unit, literature and commercial processors documentation was reviewed during earlier phases of the project. An extensive review on commercial processors was also done in August to model a realistic 4-SMT processor.
- Simulation environment. The implementation of our designs in the simulation environment involved three subtasks:
  - Training. The simulator implementation was one of the most time consuming parts of the project. The simulation environment and infrastructure was new, and a training period was necessary prior to the implementation of any new feature.
  - Implementation of conventional instruction cache hierarchies on the simulation environment. Previous works assumed that the instruction cache was perfect. Thus, we needed to add the instruction cache hierarchy and a realistic fetch function for both single thread and multithread execution.
  - Implementation of the iLP-NUCA. Adapting the structure to the instructions hierarchy was not a straightforward task, as instruction fetches have different characteristics than data requests that have to be taken into account in the iLP-NUCA structure. Besides we extended the model with a new transport network that works not only for instructions, but also for data.
- Evaluation. Evaluation of the goodness of a new design implies run simulations and analyze the outcoming results. In addition to evaluate our designs, we develop a structure to automatize the creation of simulation files and parsing results, and that could be easily adapted for others simulation environments.

The realization of this Master's Thesis makes it possible to deliver a poster presentation in an international conference, a paper submission and acceptance to a national conference, and a paper submission to a international conference.

The approximate effort invested in this Master's Thesis is 980 hours.

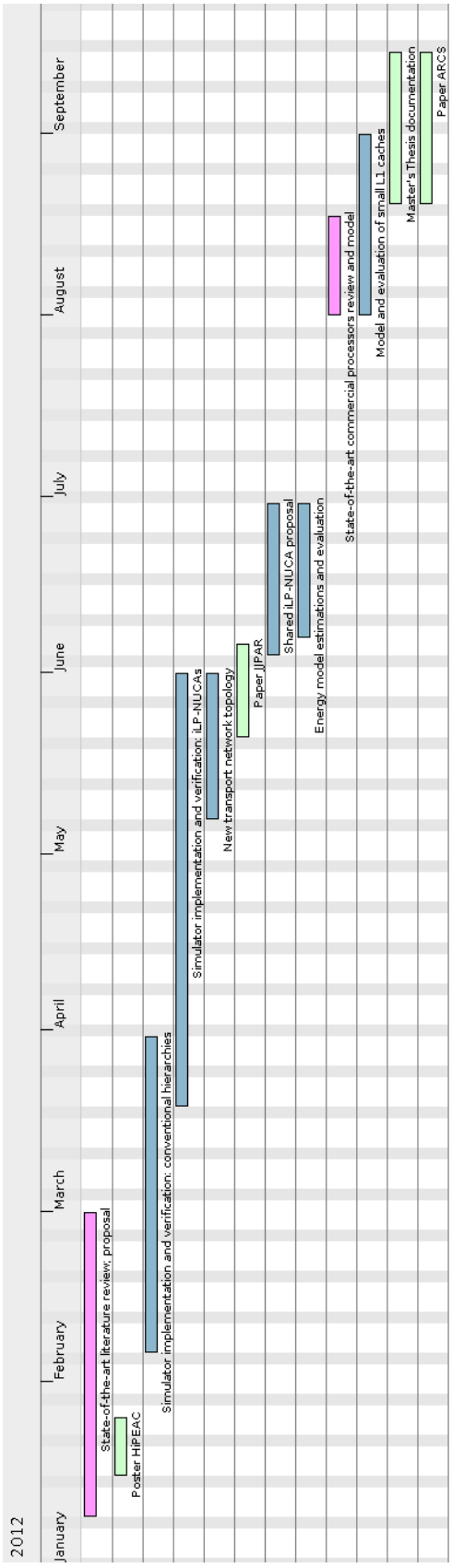


Figure A.1: Schedule and tasks of the Master's Thesis.

Appendix B | Simulation environment and methodology

In this appendix we will present in a bit more detailed the simulator infrastructure, the methodology we followed, and our scientific workflow.

B.1 SMTScalar

SMTScalar is a cycle-accurate execution-based simulator based on SimpleScalar 3.0d for Alpha ISA [4]. *SimpleScalar* was heavily extended to support detailed microarchitectural models, highly configurable memory hierarchies, and simultaneous multi-threading execution for previous LP-NUCA works [48, 47]. We extended *SMTScalar* to add instruction cache hierarchies.

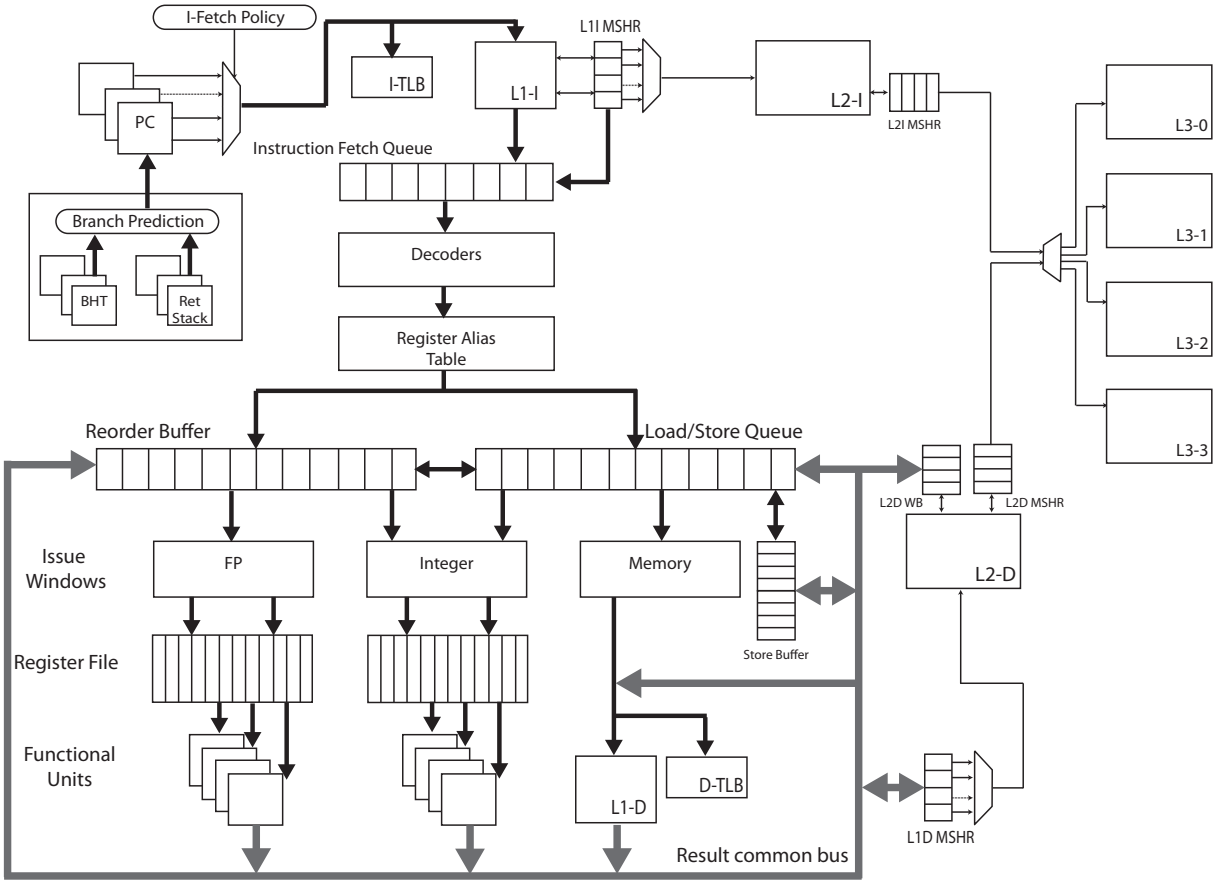


Figure B.1: Organization overview of the baseline simulated processor with a conventional three level cache hierarchy. For the sake of clarity, the L3 MSHR and the memory controller are not shown.

Figure B.1 shows the main blocks of the simulated processor. The upper part represents the fetch unit. Unlike previous works we model a instruction cache hierarchy. It supports two types of instruction hierarchies: conventional multibanked with 2 or 3 levels, plus ILP-NUCA/LP-NUCA caches. The structures inside the processor are shared by all threads, but the program counters and the branch predictor. In ech cycle up to 4 instructions can be fetched, and we can fetch instructions from two different threads. In order to allow two threads to fetch each cycle, our L1 instruction cache is multiported (2 read/write ports). Fetchs are selected following the icount policy [51]. Fetched instructions go to the Instruction Fetch Queue (IFQ). The IFQ acts as a prefetch buffer, it has 16 entries, and it is shared by all the threads. The size of this queue is critical inside the fetch unit. If the IFQ is too big, we might fetch many instructions from wrong paths (mispredicted branches) and that would imply to throw away this instructions and access the L1-I cache again. Also the area and the power dissipation should be taken into account. After running some experiments we decided that the optimal size for the IFQ was 16.

B.1.1 The microarchitectural model

Modeling in detail the microarchitectural parameters of a system involves two steps. First, we need to analyze the state–of–the–art regarding commercial systems. This information is often hard to find, as many vendors do not publish many details of their designs. Second, we need to evaluate the model. Once the parameters are fixed, we need to evaluate the system to detect possible errors in the dimensioning. An error in the the dimension of one component might cause a bottleneck in the system and cause the results to be misleading.

Our system parameters are based in state–of–the–art commercial embedded processors such as IBM/LSI PowerPC 476FP, NetLogic XLP864, and Freescale QorIQ AMP T2080 [35, 18, 7]. All of them are high–performance low–power systems oriented to embedded applications.

We run experiments adjusting different parameters such as Instruction Fetch Queue size, Issue Windows size, number of Functional Units, Reorder Buffer and Load Store Queue number of entries, among others, in order to come up with a model as realistic as possible and bottleneck free.

B.2 Simulation methodology

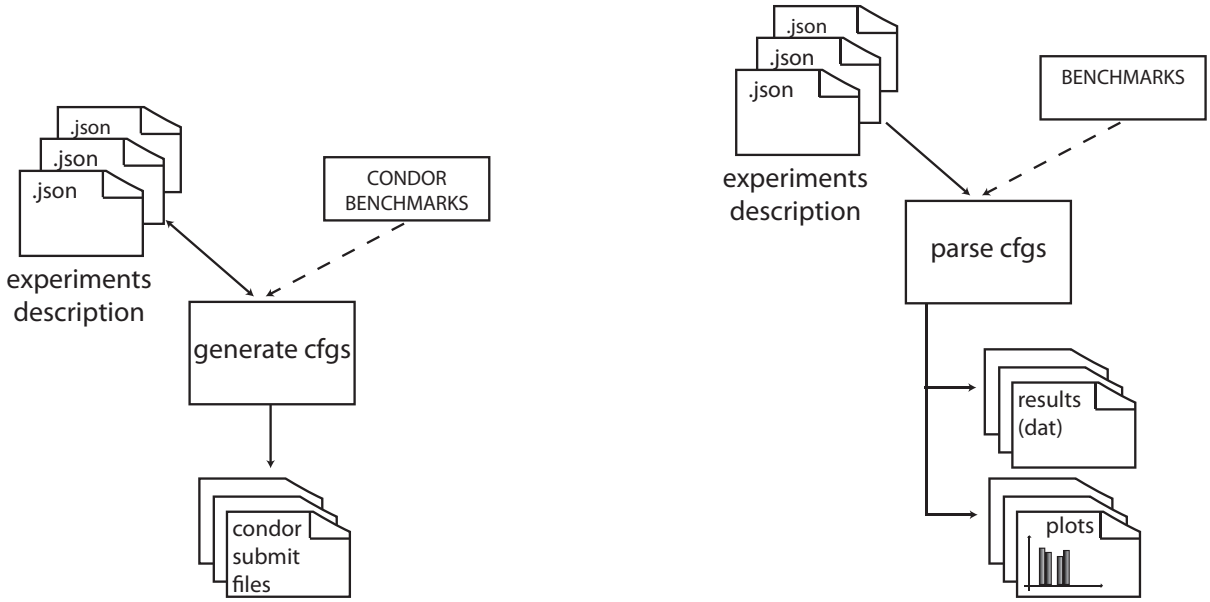
SMTScalar is a simulation environment utilized by several members of the research group. From early phases of this project one of our objectives was to share a common infrastructure with as many people as possible. Thus, we developed a simulation methodology or scientific workflow that could easily be adapted to different simulation environments.

Figure B.2 shows our scientific workflow. Our key ideas are to decouple the functionality and to automatize the simulation process. One of the main problems researchers like us suffer is the repeatability of the experiments/results. In our experiments we guarantee this by 1) defining experiments to run in files[1] (Figure B.2a), and 2) keeping the identifier of the simulator within the results files. In any given time a experiment can be repeated with the same configuration parameters and simulator file, in a fast and accurate way.

With the experiments descriptions we generate the scripts to run them depending on the system where they are going to be simulated, the benchmarks we want to test, and the job queue system.

Once the experiments are ready, Figure B.2b, we call the appropiate parser file to extract the data we are interested. As parsing the results files is not an immediate task (it can take several minutes depending on the amount of jobs simulated) we cache the relevant information in comma

[1]We use JSON, a text-based open standard designed for human-readable data interchange: www.json.org



(a) Process to generate execution files.

(b) Process to parse the results files.

Figure B.2: Simulation methodology workflow.

separated values files that allow for fast dumping.

The next step in our standardization process is to unify the the relevant information representation. We already made some progress and members of the group are able to interchange different tools just doing little adjustments.

Our future goal is to adopt a standard representation of all the files and programs involved so that different simulation environments can share the common infrastructure.

Appendix C | SPEC CPU2006 characterization: instruction cache requirements

This chapter presents the characterization of SPEC CPU2006 regarding instruction cache requirements.

C.1 Cache size and associativity implications in performance

We present for the 28 benchmarks of SPEC CPU2006 considered the impact of the first level instruction cache (L1-I) size and associativity.

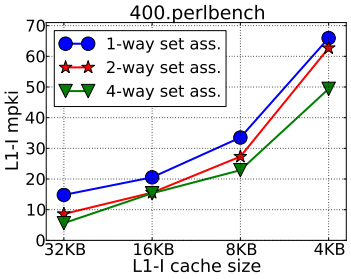
We modeled the system presented in Chapter 3 with a first level data cache that always hit. We consider four cache sizes: 32KB, 16KB, 8KB, and 4KB, and three different associativities: 1 (direct-mapped), 2, and 4. We present in the following sections the misses per k-instruction (mpki) of the first level instruction cache and IPC (instructions per cycle) for the different cache sizes. In both cases each line corresponds to the evolution of a given associativity.

C.1.1 L1-I misses per k-instruction

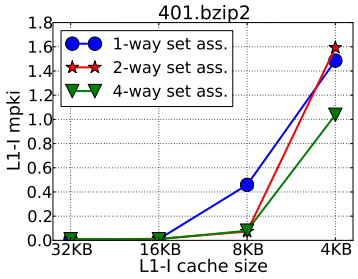
Figure C.1 shows the L1-I mpki for all the SPEC CPU2006 benchmarks considered for several sizes. Each line represents the evolution of a given associativity. Please observe that the y axis are scaled different depending on the figure for a better understanding.

From these graphs we can observe different behaviors. Some of the benchmarks, such as 458.sjeng or 465.tonto, experiment a growing L1-I mpki the very moment we reduce the cache size. Others, such as 416.gamess, experiment a big increase when the cache size is reduced to 8KB, and some others do when it is reduced to 4KB (for example 410.bwaves). There is a group of benchmarks whose working sets are small enough to fit in a 4KB direct mapped cache. Thus, they do not present a significant variation in the L1-I mpki, which is always around 0. In general associativity does not influence drastically the behavior, but in 447.dealII, where a direct mapped cache of any size increases the L1-I mpki. 403.gcc and 429.mcf also benefit from higher associative caches, specially for 16KB cache size.

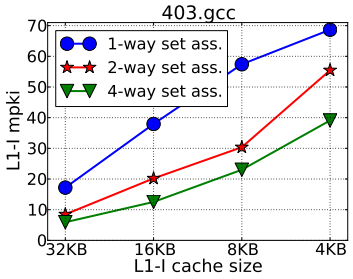
These results show that, in general, instruction cache requirements of SPEC CPU2006 benchmarks are low, but some exceptions (such as 445.gobmk, 444.namd, 458.sjeng, or 482.sphinx3, among others), and a 32KB 4–way set associative cache is able to capture in most of the cases the instruction footprint of the application.



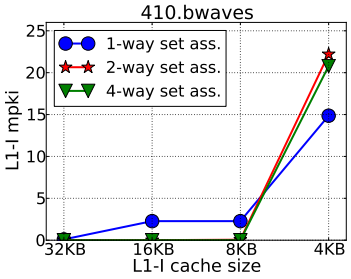
(C.1.1) L1-I mpki, 400.perlbench



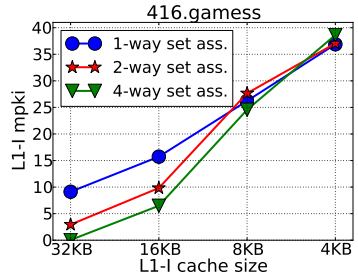
(C.1.2) L1-I mpki, 401.bzip2



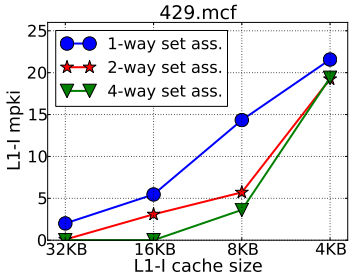
(C.1.3) L1-I mpki, 403.gcc



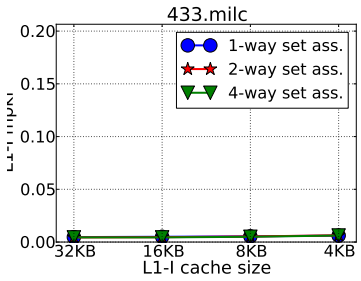
(C.1.4) L1-I mpki, 410.bwaves



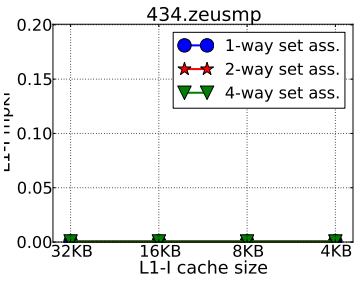
(C.1.5) L1-I mpki, 416.gamess



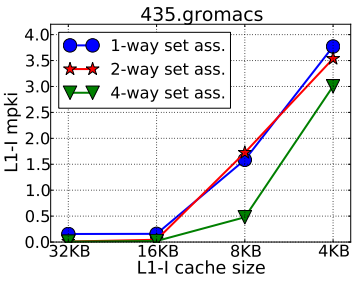
(C.1.6) L1-I mpki, 429.mcf



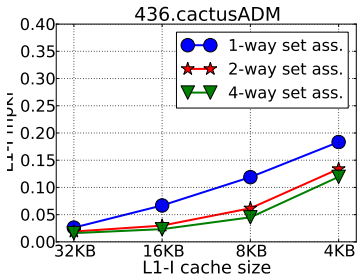
(C.1.7) L1-I mpki, 433.milc



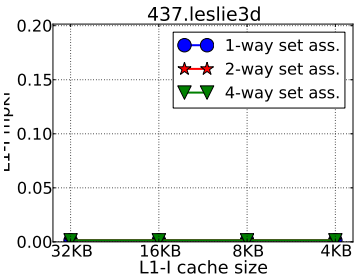
(C.1.8) L1-I mpki, 434.zeusmp



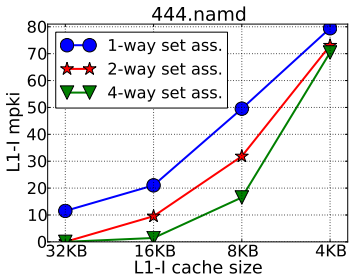
(C.1.9) L1-I mpki, 435.gromacs



(C.1.10) L1-I mpki, 436.cactusADM

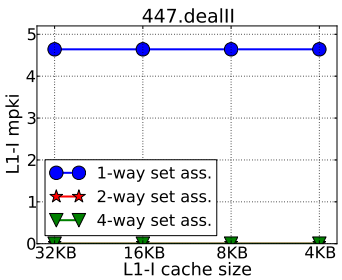


(C.1.11) L1-I mpki, 437.leslie3d

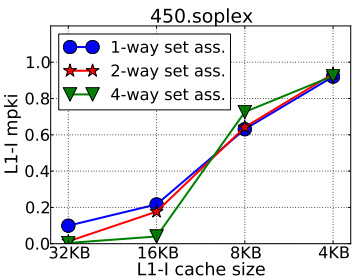


(C.1.12) L1-I mpki, 444.namd

(C.1.13) L1-I mpki, 445.gobmk

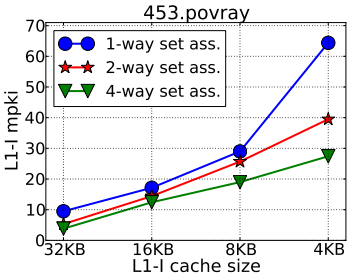


(C.1.14) L1-I mpki, 447.dealII

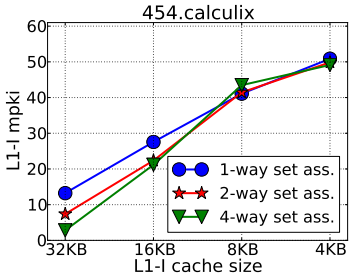


(C.1.15) L1-I mpki, 450.soplex

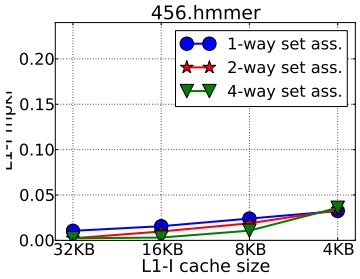
Figure C.1: L1-I mpki for several cache sizes and associativities, SPEC CPU2006.



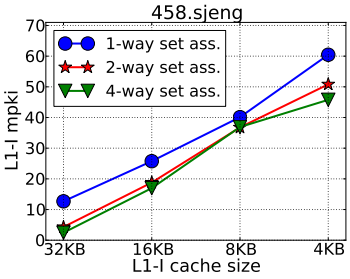
(C.1.16) L1-I mpki, 453.povray



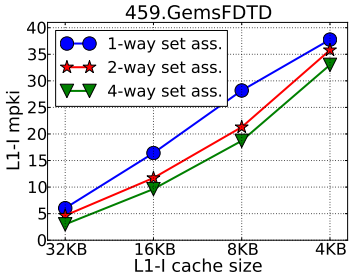
(C.1.17) L1-I mpki, 454.calculix



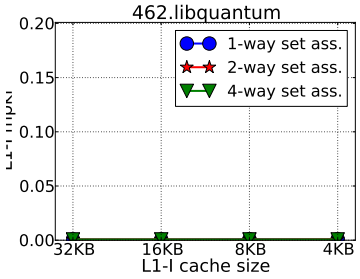
(C.1.18) L1-I mpki, 456.hmmer



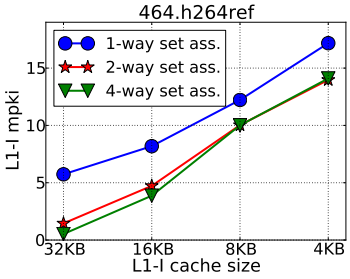
(C.1.19) L1-I mpki, 458.sjeng



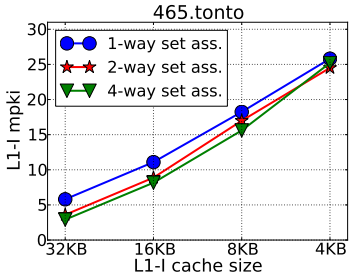
(C.1.20) L1-I mpki, 459.GemsFDTD



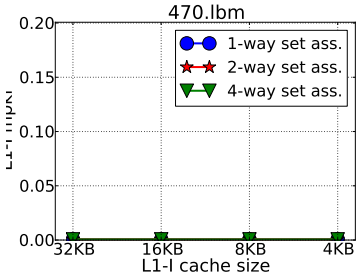
(C.1.21) L1-I mpki, 462.libquantum



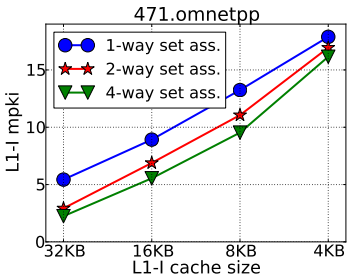
(C.1.22) L1-I mpki, 464.h264ref



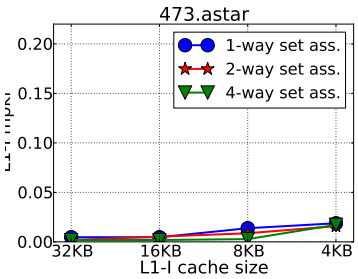
(C.1.23) L1-I mpki, 465.tonto



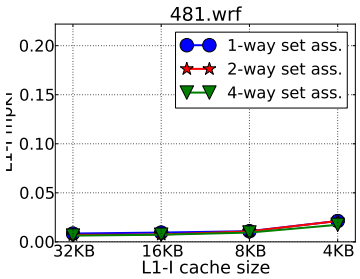
(C.1.24) L1-I mpki, 470.lbm



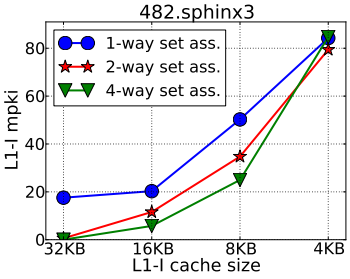
(C.1.25) L1-I mpki, 471.omnetpp



(C.1.26) L1-I mpki, 473.astar



(C.1.27) L1-I mpki, 481.wrf



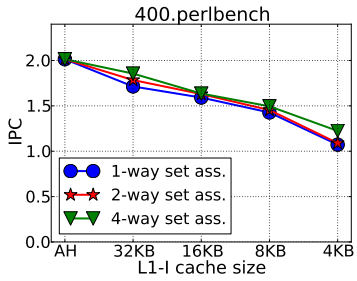
(C.1.28) L1-I mpki, 482.sphinx3

Figure C.1: L1-I mpki for several cache sizes and associativities, SPEC CPU2006.

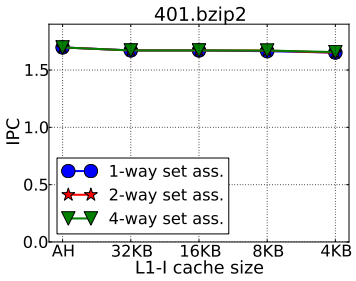
C.1.2 Performance evaluation: IPC

Figure C.2 shows the IPC for all the SPEC CPU2006 benchmarks considered for several sizes. Each line represents the evolution of a given associativity. AH represents a L1-I cache that always hits. This value gives us an upper bound of how much performance degradation, in comparison with a ideal cache, each application might suffer.

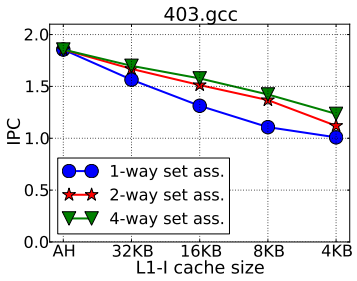
From the figures we can see that, in general, increases in the L1-I mpki implies performance degradations. We can also observe that 32KB 4–way set associative caches are in general very close to the ideal performance.



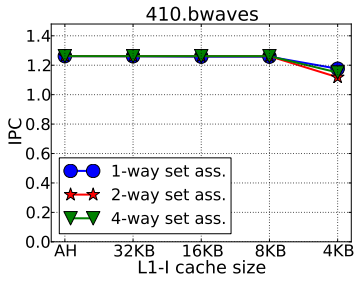
(C.2.1) IPC, 400.perlbench



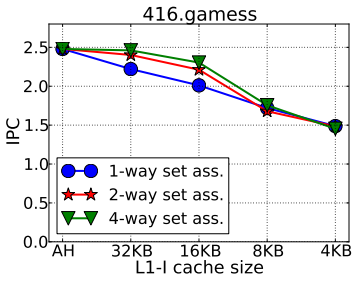
(C.2.2) IPC, 401.bzip2



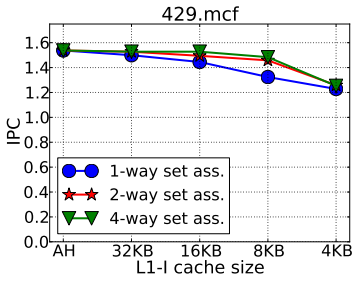
(C.2.3) IPC, 403.gcc



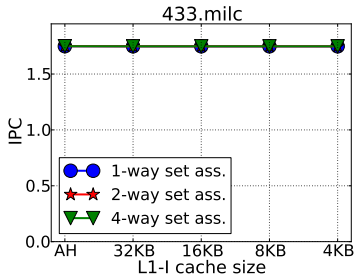
(C.2.4) IPC, 410.bwaves



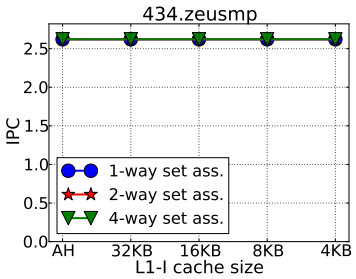
(C.2.5) IPC, 416.gamess



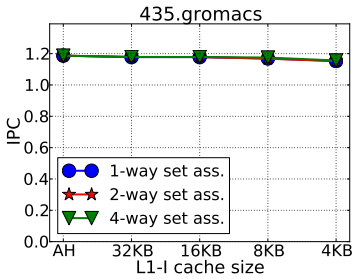
(C.2.6) IPC, 429.mcf



(C.2.7) IPC, 433.milc

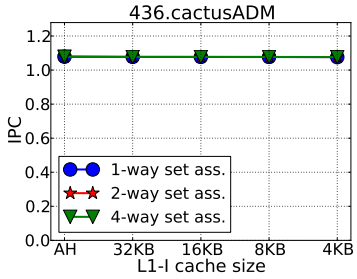


(C.2.8) IPC, 434.zeusmp

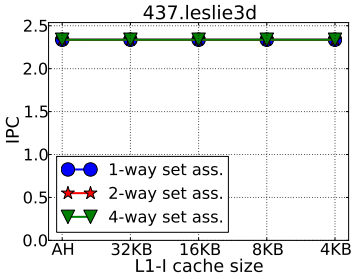


(C.2.9) IPC, 435.gromacs

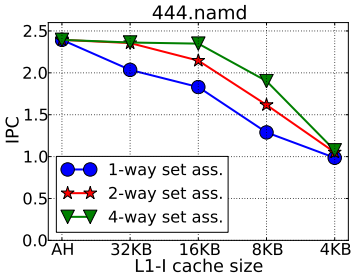
Figure C.2: IPC for several cache sizes and associativities, SPEC CPU2006.



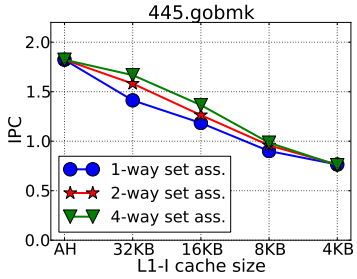
(C.2.10) IPC, 436.cactusADM



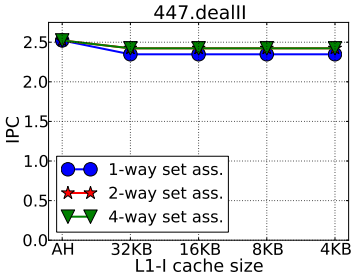
(C.2.11) IPC, 437.leslie3d



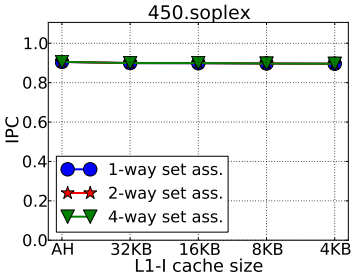
(C.2.12) IPC, 444.namd



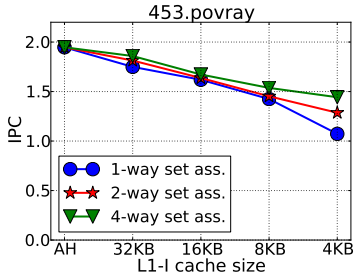
(C.2.13) IPC, 445.gobmk



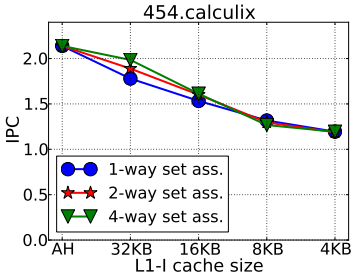
(C.2.14) IPC, 447.dealII



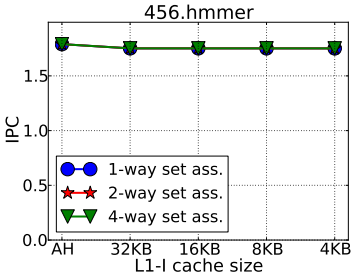
(C.2.15) IPC, 450.soplex



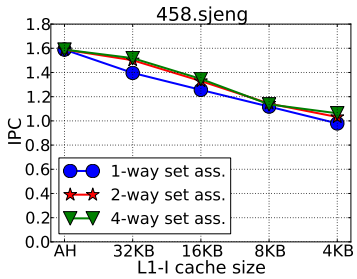
(C.2.16) IPC, 453.povray



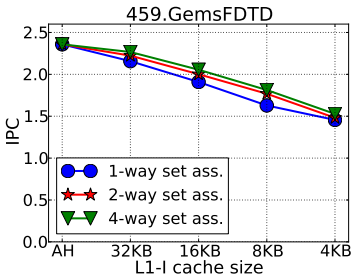
(C.2.17) IPC, 454.calculix



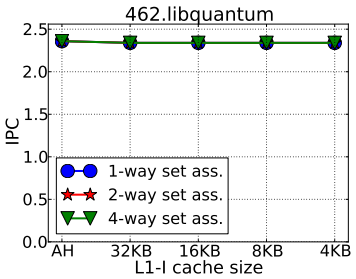
(C.2.18) IPC, 456.hmmer



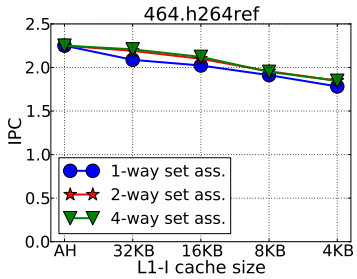
(C.2.19) IPC, 458.sjeng



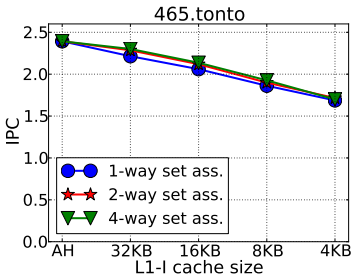
(C.2.20) IPC, 459.GemsFDTD



(C.2.21) IPC, 462.libquantum



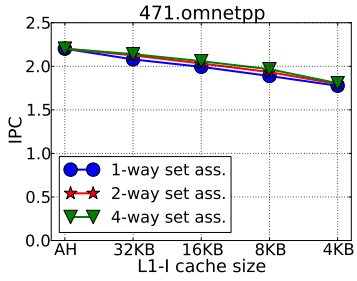
(C.2.22) IPC, 464.h264ref



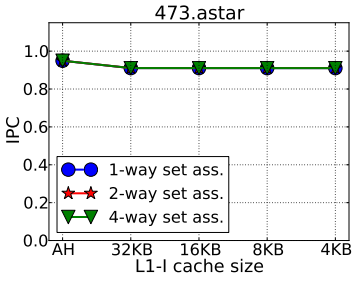
(C.2.23) IPC, 465.tonto

(C.2.24) IPC, 470.lbm

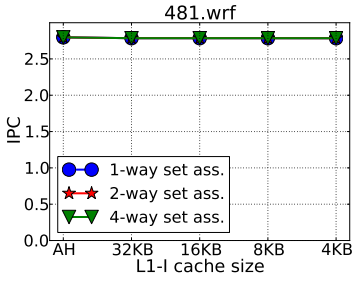
Figure C.2: IPC for several cache sizes and associativities, SPEC CPU2006.



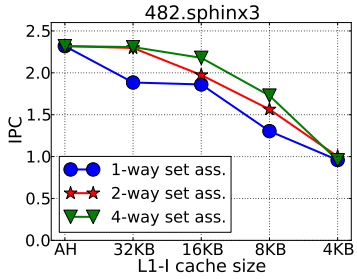
(C.2.25) IPC, 471.omnetpp



(C.2.26) IPC, 473.astar



(C.2.27) IPC, 481.wrf



(C.2.28) IPC, 482.sphinx3

Figure C.2: IPC for several cache sizes and associativities, SPEC CPU2006.

Appendix D | Shared iLP-NUCA designs

In this appendix we will introduce our ongoing work on shared iLP-NUCA designs.

Although dedicated (separated) caches offer higher performance, they have also higher energy and area requirements, both critical parameters in the embedded systems domain. Thus, last level caches (LLC) are usually shared for instructions and data caches.

Sharing the iLP-NUCA, however, is not straightforward. iLP-NUCA relies on very specialized networks-in-cache, naming search, transport, and replacement. One of the main advantages of the structure is that messages are implicitly routed, as they only have one possible destination (the root-tile).

The critical path on iLP-NUCA tile consists on a hit in the cache followed by the transport operation to the neighbor tile. This operation fits in a cycle time because the network operation can be simplify, among other things, by deleting the routing stage. Adding a second root-tile to the structure means that in a complete shared design messages need to decide which path to follow (we need a routing mechanism). Other important issue is the connection wires length, which might become too long, increasing the delay.

On the other hand iLP-NUCA networks-in-cache offer also advantages. Its structure would allow for a hybrid design, where some tiles would be private for instructions, some others for data, and some would be shared. In this way we would be able to restrict the amount of space data and instructions occupy and guarantee that, for example, data demanding applications pollute the cache and evict instruction blocks to the next level.

A possible way to implement these ideas is to modify the replacement operation and restrict the replacement through selected links. Besides we should include a routing mechanism.

Figures D.1, D.2, and D.3 show a possible design for the networks-in-cache topologies that takes advantage of these ideas.

We want to evaluate this new structure, and right now we are implementing this design in our simulation environment.

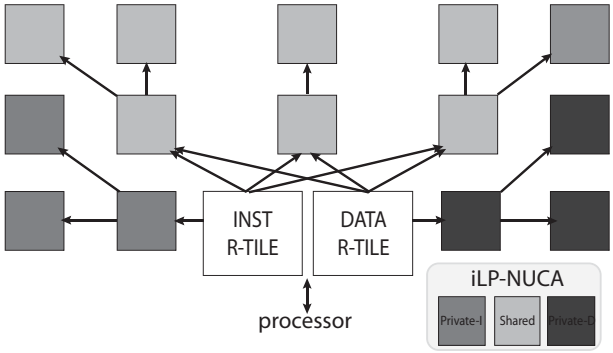


Figure D.1: Search network topology for shared iLP-NUCA.

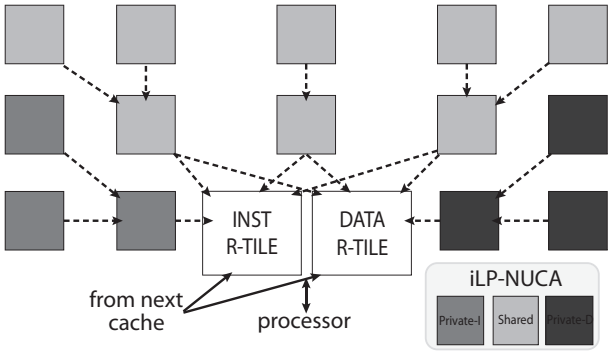


Figure D.2: Transport network topology for shared iLP-NUCA.

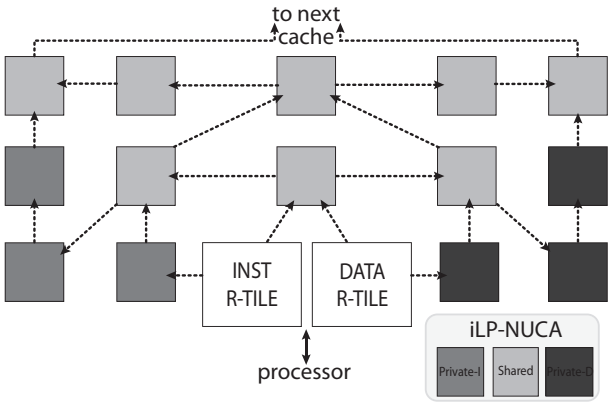


Figure D.3: Replacement network topology for shared iLP-NUCA.

Appendix E | ILP-NUCA: Cache de Instrucciones Teselada para Procesadores Empotrados

This appendix includes the paper *ILP-NUCA: Cache de Instrucciones Teselada para Procesadores Empotrados* by Alexandra Ferrerón, Marta Ortín, Darío Suárez, Jesús Alastruey, and Víctor Viñals.

This paper was submitted and accepted into the *XXIII Jornadas de Paralelismo*. It will be presented in Elche in September 2012.