Final Report:

MIPS R12000 Processor Performance Evaluation by SPEC2000 Benchmarks and Performance Counters.

Author: Paolo D'Alberto
Date: Jan 27 2000

Abstract.

The goal of our investigation is to achieve a quantitative measure of the architecture *utilization* for a set of applications: such a measure can be used as feedback to confirm the design properties of the super-scalar microprocessor but also to understand the impact of design choices of single modules over the system organization. We collected experimental results from SPEC 2000 benchmark suit using the performance hardware counters available on R10K/R12K microprocessors. Several events have been collected offering, even if partially, a very powerful mean to investigate how hardware enhancements, such as branch prediction and cache replacement policy, are effective.
We have found that in general branch prediction is effective but less than expected achieving in average 85% correct prediction. The data pre-fetch is not effective; 15% of the times, useful data are fetched (which are not already in cache). Memory hierarchy is bottleneck for performance.

Introduction.

We focus our attention on the interpretation of performance measurements. The current software/hardware technologies offer the chance to collect several performance measurements using different tools and different approaches: *profiling, hardware counters* [3] and *simulation.*
Profiling is mostly used to determine hot spots of an application. The code is instrumented, sometimes introducing extra code, so that execution time (or other features) of procedures, loops or simple block of code can be measured and updated every time the execution reaches the interested set of instructions. The approach is very often invasive since the application itself may be modified during compilation time. Often, the measure is not exact: the execution of the application is checked only after an interval of time randomly determined, or after a fixed number of cycles (usually a prime number), statistics are collected. The inserted code may have the form of a process running concurrently to the application, halting its execution and collecting/computing related information, but competing for the same resources.
Hardware counters collect events at processor level, it is like profiling but the extra code is already available in hardware. R10K/12K processors allow 32 events statistics. Each of them represents a feature of the architecture, i.e. the number of cycles or the number of

misses at the first level of cache. Hardware counters do not describe completely the performance of a microprocessor, far from it, but permit to collect exactly the numbers for a few and important events. Furthermore, not all 32 events can be counted exactly at the same time, respecting a kind of Heisenberg's indetermination principle applied to microprocessors [1,2,3].
Simulation is a common approach to obtain very detailed information about application performance. Given an executable and a model of the sub/system, each instruction is evaluated with respect to the model; the current status of the pair application-architecture is determined. More accurate is the model, more precise is the information collected. **Flexibility**: different modules simulating a particular device can be plugged in. The main disadvantage is the execution time to obtain the experimental results. It is very common to achieve a slowdown of two orders of magnitude (50-100 times slower). This can be a problem when the application requires the execution of some thousands billions instructions.
The applications are from SPEC 2000, which is a collection of benchmarks chosen so that modern architectures can be tested more thoroughly, i.e. the space requirements are more demanding exploiting the memory hierarchy organization and different applications may exploit the accuracy of the hardware branch prediction. We used in another investigation simulation tools to collect information about the type of data cache misses (*capacity, compulsory* and *conflicts*) inherent characteristic of the pair application-cache. Indeed, the measure can be easily obtained by interchanging different model of data cache. We can say that the results exploit the capacity nature of the data misses. Therefore the most advantageous way to improve cache performance it should be to increase the cache capacity. Anyway, we used hardware counters to measure performance of the cache hierarchy. The average data miss ratio is low, 7% for the first level and 2% for the secondary level of cache. The average instruction miss ration is 0.1% and it can be considered negligible. We do describe the relationship between processor utilization and data misses; we show that the memory hierarchy is the bottleneck for performance, in fact, reporting a interesting comparison between CPI (Cycle Per Instruction) and data miss ratio cross the benchmarks.
The report is organized as follows: we introduce a description of the super-scalar microprocessor R12K, the events we are able to count through hardware counters and how we can count them; we introduce the benchmarks giving a brief description of each one of them; we introduce the experimental results from performance hardware counters and an interpretation of them and eventually our conclusions.

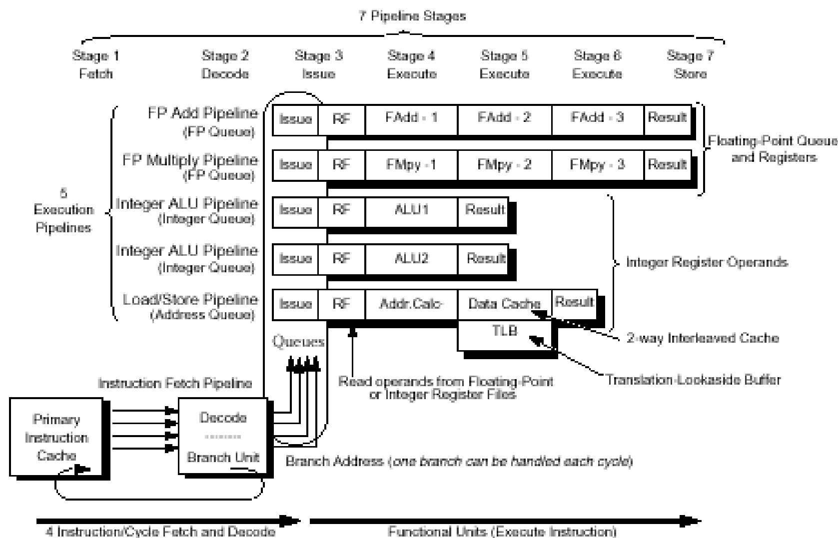
R10K Microprocessor.

A brief description and explanation of the microprocessor is the goal of this section. We introduce the main features of processor R10000 [1] and, by difference, the processor R12000 [2].

- It implements the 64-bit MIPS IV instruction set architecture.
- It decodes four instructions per cycle and appends them to three *different instruction queues.*
- It has five *execution queues*: 2 are integer queues, 2 are single/double floating point queues and 1 is an address queue.
- It uses ***Dynamic Instruction Scheduling*** and ***Out of Order Execution.***

- It uses ***Speculative Instruction Issue.***
- It uses precise exception model.
- It uses non blocking caches.
- It has 32KB 2-way interleaved associative on chip data and instruction cache.

Taking from [1] a picture summarizing the R10K pipelines:



The integer queue issues instructions to two **ALU**s, the floating point queue issues instructions to the floating point adder and multiplier and the address queue issues to the **Load/Store Unit.**

R10000 Pipeline Stages

- **Stage 1:** Four instructions are fetched per cycle independently from their alignment in the 16-word cache line, unless they cross the line border. Then the four instructions are stored contiguously in a 4-word *Instruction Register.*
- **Stage 2:** The four instructions are decoded, dependency among instructions is determined and renaming is performed. Integer register renaming is performed in parallel with floating point register renaming and such information is stored in **physical register busy table**. Only one branch is executed at this stage, if any other branch instruction is in the instruction register will be decode in the following cycle. The *Branch Unit* determines the following up address. If the condition of the branch is based on the result of not yet to complete instruction, the address is guessed.
- **Stage 3:** Decoded instructions are inserted in the proper queue and their executions are started.
- **Stage 4-6** The execution of instructions is carried on. When the operands of an instruction are ready, the execution of the instruction can start:
  - Integer addition, logic operations and shift operation take one cycle.
  - Integer multiplication and division take more than one cycle.
  - Floating point addition and multiplication have 2-4 cycles latency and internal bypass is available for multiply-add instruction.
  - Floating point division and square root take more than 3 cycles.

The block Diagram of R10K is described in the next figure:

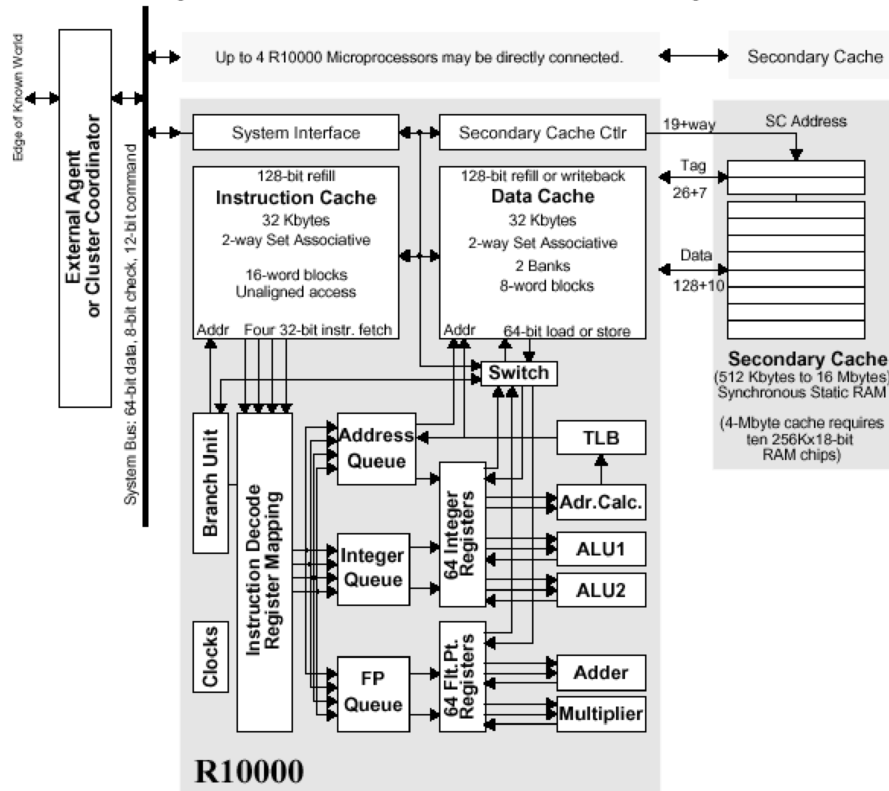


Figure 1-5 Block Diagram of the R10000 Processor

Branch Unit

Only one branch per cycle is allowed, but up to 4 deep conditional branches can be executed speculatively. Two bits prediction is associated with every branch, describing the likeness of been taken or not (strongly taken, weakly taken, weakly not taken and strongly not taken). To revert the wrongly taken branches a 4-quadword-branch-resume buffer is used. Branch prediction offers the chance to fetch instructions that will be most probably executed in the future, the draw back is when a branch is miss-predicted: the pipelines must be flushed and the execution must roll back.

Integer ALU pipeline structure

The Integer pipeline has a 16-entry instruction queue which issues instructions *dynamically,* i.e. the issue order may be different from the queue insertion order. ALU1 contains an arithmetic-logic unit, a shifter and an integer branch comparator. ALU2 contains an arithmetic-logic unit, integer multiplier and an integer divider. When a division is issued then ALU2 is busy as long to the division completion (up to 67 cycles). It is available a 64 bit 64 locations register file, implementing a 32 location logical register file. *Register renaming* it is performed when instructions are decoded; register names are resolved and associated with real registers.

Floating Point pipeline structure.

It has a 16 entry queue with dynamic issue, it has a multiply unit with 2+1 cycles (2 cycles latency + 1 cycle repeat, repeat cycle is the minimum number of cycles to issue another instruction to the same unit) latency and an adder unit with 2+1 cycle latency as well. The multiplier issue queue is shared with a divide unit and a square root unit. Indeed, the completion logic is shared as well; but the execution is performed in parallel (the square root for doubles has up to 52 cycle latency).

Instruction Decode and Rename Unit.

Up to four instructions are decoded at any cycle. It is available a 32-entry active list of all instructions within the pipeline, there is a 33-word-by-6-bit mapping table to resolve integer register renaming, there is a 32-word-by-6-bit mapping table to solve floating point register renaming.

Address Queue: Load/Store Unit

The address queue is a 16-location list organized as circular FIFO; the issue order and insertion order is the same, respecting the program original order. Instructions stay in the queue as long as the unit completes the execution of the instruction; the instruction is graduated. The queue has three issue ports:

1. Each instruction is issued to the *address calculation unit* once: it is a 2 stages unit where the instruction address is computed and translated in the TLB. Even if the cache array is not available the tag check can be performed.
2. An instruction can be re-issued to the data cache. The instruction starts with a tag check; if the data is available and no dependencies exist, the instruction is marked *done*; if the data is not available the instruction waits and a refill operation starts.
3. Store instruction is issued to the data cache. Only one store can graduate per cycle.

A load instruction has 2-3 cycles latency (integer-floating point) and one cycle repeat, if the data is available in the cache. A store has one cycle repeat.

I-Cache

It is a 32KB cache organized into 16-word line; it is a 2-way associative using LRU policy. It reads four consecutive instructions per cycle anywhere in the line but not crossing the line border. A pre-decode of instructions is performed at this stage discriminating the nature of the instruction (i.e. integer or floating point instruction). The cache is non-blocking, i.e. a miss do not block the successive request to the cache that can be fulfilled.

D-Cache

It is a 32KB cache made of two interleaved array and organized into 8-word line. It is a 2-way associative cache using LRU policy and write back (i.e. any store or write performed onto it is not performed to the next level unless necessary).

Secondary Cache

The size of the cache ranges from 512KB to 16MB, it is a 2-way associative organized with 7/15-word line. The LRU policy is implemented and the access tag check is not performed in parallel (way prediction). A D-cache line, 8-words, is refilled in two cycles and

I-cache line in 4 cycles due to the clock difference between modules. The latency to access the secondary cache is between 6 and 12 cycles; fluctuations are possible due to the synchronization of the devices.

R12K Processor

We are going to introduce the architecture of microprocessor R12K by comparison from the architecture of R10K microprocessor [2]. Let us introduce the pipeline structure and composition:

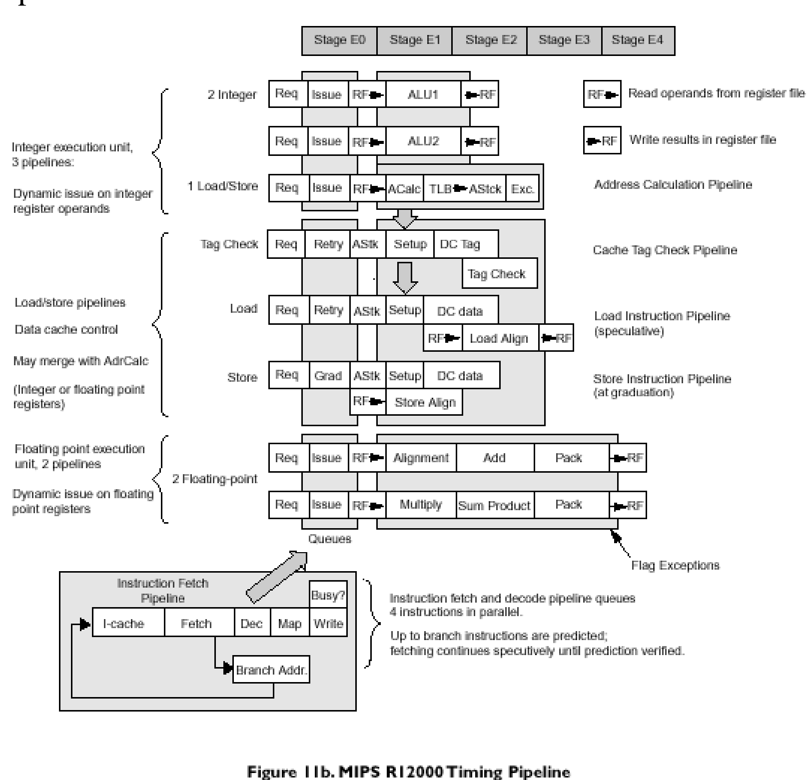


Figure 11b. MIPS R12000 Timing Pipeline

In this Figure we can see in more details the three pipelines dedicated to the cache control, Tag Check, Load and Store in the Address queue. In the Figure in the next page we can see the organization of the microprocessor. The prediction table has been increased by four times, from 512 entries to 2048 entries, improving the accuracy. A 32 entry *Branch Target Address Cache* has been introduced, so that whenever the target address is found in the cache, the address computation can be avoided, and a cycle can be saved. The list of active instructions has been increased, from 32 to 48 entries, increasing the chance to find more *useful* instructions to execute.

A simplification of the design is the address computation instructions, load/store instructions are sent to the integer instruction queue and then to the address queue.

A further improvement can be found in the implementation of the *Most Recently Used* policy for the way prediction on the secondary cache. Tag check is performed on each block of the secondary cache sequentially, but the order which array will be checked first it is guessed before every access.

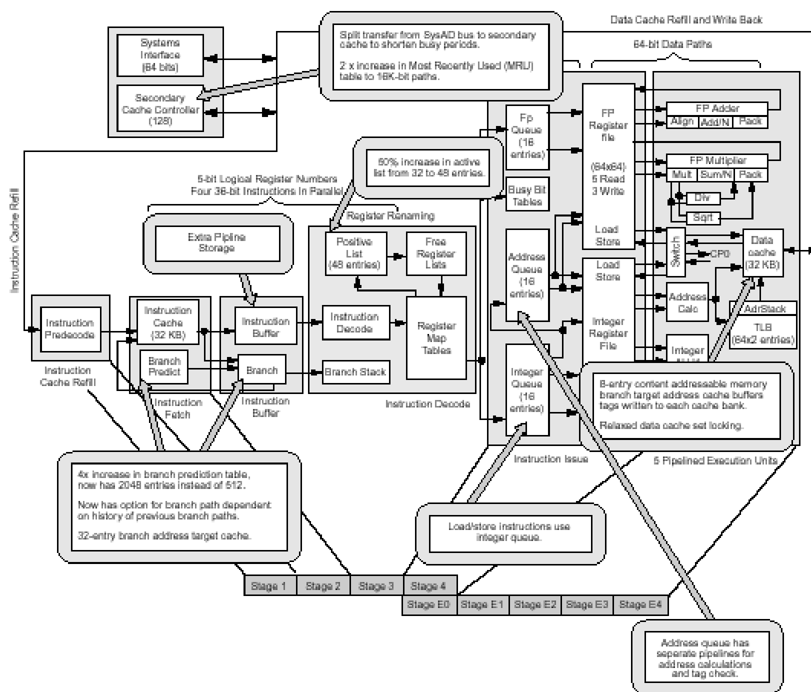


Figure 11a. Summary of Changes Between MIPS R10000 and MIPS R12000 Microprocessor Architectures

Hardware Counter

The R12K microprocessor support two performance counters to count 32 events, 16 events for each register. It follows a short description of each one of them (taken from the manual page):

- **Event 0**: Incremented on each clock cycle.
- **Event 1**: Incremented by the total number of instructions decoded on the previous cycle. The count reflects the overhead due to incorrectly speculated branches and exception processing.
- **Event 2**: Incremented when a load instruction was decoded on the previous cycle.
- **Event 3**: Incremented if a store instruction was decoded on the previous cycle. Store conditionals are included in this count.
- **Event 4**: Incremented on each cycle by the number of currently valid entries in the Miss Handling Table (MHT). The MHT has five entries. Four entries are used for internally generated accesses; the fifth entry is reserved for externally generated events.
- **Event 5**: Incremented when a store-conditional instruction fails. A failed store-conditional instruction eventually graduates.
- **Event 6**: Incremented when a branch has been predicted correctly and when it has been incorrectly predicted.
- **Event7**: Incremented on each cycle that the data for a quad-word is written back from secondary cache to the system interface unit.
- **Event 8**: Incremented on the cycle following the correction of a single-bit error in a quad-word read from the secondary cache data array.

- **Event 9**: Incremented one cycle after an instruction fetch request is entered into the Miss Handling Table.
- **Event 10**:Incremented the cycle after a refill request is sent to the system interface module of the CPU.
- **Event 11**: Incremented when the secondary cache control begins to retry an access because it hit in the unpredicted way, provided the access that initiated the access was an instruction fetch.
- **Event 12**: Incremented on the cycle after an intervention is entered into the MHT, provided that the intervention is not an invalidated type.
- **Event 13**: Incremented on the cycle after an intervention is entered into the MHT, provided that the intervention is an invalidate type.
- **Event 14**: Incremented on the cycle after either ALU1, ALU2, FPU1, or FPU2 marks an instruction as done.
- **Event 15**: Incremented by the number of instructions that were graduated on the previous cycle. Integer multiplication and division instructions each count two graduated instructions because they occupy two entries in the active list.
- **Event 16**: Incremented on the cycle after a prefetch instruction does its tag-check, regardless of whether a data cache line refill is initiated.
- **Event 17**: Incremented on the cycle after a prefetch instruction does its tag-check and a refill of the corresponding data cache line is initiated.
- **Event18**: Incremented by the number of loads that graduated on the previous cycle. Prefetch instructions are included in this count.
- **Event 19**: Incremented on the cycle after a store graduates. Only one store can graduate per cycle. Store conditionals are included in this count.
- **Event 20**: Incremented on the cycle following the graduation of a store-conditional instruction. Both failed and successful store-conditional instructions re included in this count.
- **Event 21**: Incremented by the number of floating-point instructions that graduated in the previous cycle.
- **Event 22**: Incremented on each cycle that a quadword of data is valid and is written from primary data cache to secondary cache.
- **Event 23**: Incremented on the cycle after the translation look-a-side buffer (TLB) miss handler is invoked.
- **Event 24**: Incremented on the cycle after a branch is **restored** because it was mispredicted.
- **Event 25**: Incremented one cycle after a request is entered into the SCTP logic, provided that the request was initially targeted at the primary data cache. Such requests fall into three categories:
    1. Primary data cache misses.
    2. Requests to change the state of secondary and primary data cache lines from clean to dirty.
    3. Requests initiated by cache operation instructions.
- **Event 26**: Incremented the cycle after a refill request is sent to the system interface module of the CPU.

- **Event 27**: Incremented when the secondary cache control begins to retry an access because it hits in the unpredicted way. The counter is incremented only if access that initiated the access was not an instruction fetch.
- **Event 28**: Set on the cycle after an external intervention is determined to have hit in the secondary cache. The value of the event is equal to the state of the secondary cache line that was hit. Setting a performance control register to select this event has a special effect on the conditional counting behavior. If event 28 or 29 is selected, the sense of the "Negated conditional counting" bit is inverted. See the description of conditional counting for details.

  The values are:
  00 Invalid, ho hit detected
  01 Clean, shared
  10 Clean, exclusive
  11 dirty, exclusive
- **Event 29**: Set on the cycle after an external invalidate request is determined to have hit in secondary cache.
- **Event 30**:Incremented on each cycle by the number of entries in the MHT waiting for a memory operation to complete. For example, once the secondary cache tags are checked and a secondary cache miss is recognized, the entry that originated the request is included in this count. It continues to be included until the last word of the refilled line is written into the secondary cache and the MHT entry is removed.
- **Event 31**: Incremented on the cycle after an update request is issued for a line in the secondary cache. If the line is in the clean state, the counter is incremented by one. If the line is in the shared state, the counter is incremented by two. The conditional counting mechanism can be used to select whether one, both, or neither of these events is chosen.

There are different approaches to collect the event mentioned. It is possible to collect two events, one from 0-15 and one from 16-31, at the same time without measure error because using two different performance register counters. A very common performance measure is miss ratio (I-Cache/D-Cache): therefore we need to collect the number of misses and the number of accesses the application performs; using event 25 and 26 we count the number of data misses and with event 18 and 19 we count the number of graduated stores and loads (memory accesses). We need four different runs of the application, one for each event. The number of misses is dependent from the workload of the system: since another can preempt the application at anytime, the caches contents may be flushed. Therefore data misses may vary. By definition, instructions graduate when they complete and all previous instructions graduate; graduated loads and store are invariant. But the ratios are not invariant. The problem is amplified when more than four events must be collected and correlation between events is important for the analysis. It is possible to get **estimation** of all events in just one run. Indeed, the execution is split in intervals of same time size. In each interval can be collected precisely two events. In a round Robin fashion and after 16 intervals, every event has been collected on a particular interval. The totals are computed from the collected results by multiplying by 16. This is not a correct measure is an estimation, since an event can be colleted when it rarely happens and not collected otherwise. If we consider the presence of other applications for measure purposes in a short period of time as a *white noise*, every interval is affected negatively in the same

way. Therefore the measure error due to interference is the same for every event. This approach offers the chance to obtain a more confident analysis when events are used for inferred measures.

Benchmarks: SPEC2000

The benchmarks suit it is composed of two sets of applications: *Floating Point Applications* and *Integer Applications*. Intuitively, the former exploits the floating point units utilizations; the latter exploit the integer point units. In this section we are going to introduce each application, a brief description and a note whether we could not collect experimental results. The following brief summary is taken from the SPEC2000 benchmarks description [4]; some of the descriptions are very *technical* but fascinating (most of them are reported as they are from the documentation, too many references to cite!).

CFP2000

- 168 – **wupwise**: "wupwise" is an acronym for "Wuppertal Wilson Fermion Solver", a program in the area of lattice gauge theory (quantum chromo-dynamics). Lattice gauge theory is a "*discretization*" of quantum chromo-dynamics, which is generally accepted to be the fundamental physical theory of strong interactions among the quarks as constituents of matter. The most time-consuming part of a numerical simulation in lattice gauge theory with Wilson fermions on the lattice is the computation of quark propagators within a chromo-dynamic background gauge field. Quark propagators are obtained by solving the inhomogeneous lattice-Dirac equation. The Wuppertal Wilson Fermion Solver (wupwise) solves the inhomogeneous lattice-Dirac equation via the BiCGStab (Bi-Conjugate-Gradient) iterative method, which has established itself as a method of choice.
- 171 – **swim**: it is a weather prediction program. The model is taken from the paper "The dynamics of finite-difference models of shallow-water equations" by Robert Sadourny.
- 172 – **mgrid:** it demonstrates the capabilities of a simple multi-grid solver computing a three dimensional potential field.
- 173 – **applu**: Solution of five coupled nonlinear PDE's, on a 3-dimensional logically structured grid, using an implicit pseudo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix. Spatial discretization of the differential operators is based on second-order accurate finite volume scheme; it insists on the strict lexicographic ordering during the solution of the regular sparse lower and upper triangular matrices.
- 177 – **mesa**: it is a free OpenGL work-alike library. Since it supports a generic frame buffer it can be configured to have no OS or window system dependencies. Any number of client programs can be written to stress FP, scalar or memory performance (or a mix).
- 178 – **galgel**: This problem is a particular case of the GAMM (Gesellschaft fuer Angewandte Mathematik und Mechanik) benchmark devoted to numerical analysis of oscillatory instability of convection in low-Prandtl-number fluids. The physical problem is the following. There is a rectangular box filled by a liquid whose Prandtl number is Pr=0.015. The aspect ratio of the cavity length/height is 4. The left and

right vertical walls are maintained at higher and lower temperatures respectively. This causes a convective motion in the liquid. When the temperature difference is relatively small the convective flow is steady. The flow looses its stability and become oscillatory when the temperature difference exceeds a certain value. The buoyancy force, which causes the convective flow, is characterized by a parameter called Grashof number. Besides all, the Grashof number (Gr) is proportional to the characteristic temperature difference (difference of the temperatures at the vertical walls in this case). The task of the GAMM benchmark is to calculate the critical value of the Grashof number, which corresponds to a bifurcation from steady to oscillatory state of the flow. Together with the critical Gr it is necessary to calculate the critical frequency (the frequency of the resulting oscillations when Gr is equal to its critical value). The critical values (critical Grashof number and critical frequency) depend on all parameters of the problem and the boundary conditions. The GAMM benchmark considers fixed values of the Prandtl number and the aspect ratio (0.015 and 4 respectively), and varies the boundary conditions. The Galerkin method requires large computer memory required to keep all coefficients of the resulting dynamic system. To avoid this some coefficients are recalculated each time when a calculation of rhs of the dynamic system is necessary, leading to a rapid increase of the required memory and cpu time when the number of the Galerkin basis functions is increased. A relatively small number of degrees of freedom makes it possible to study linear stability of steady solutions, requiring solution of an eigenvalue problem. It becomes possible with the use of the global Galerkin method, and it was successfully done for convective flows and for swirling flow in a closed cylindrical container. After linear stability analysis is completed and the bifurcation point is calculated, an asymptotic approximation of the supercritical flow is computed.

- 179 – **art**: The Adaptive Resonance Theory 2 (ART 2) neural network is used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first trained on the objects. After training is complete, the learned images are found in the scanfield image. A window corresponding to the size of the learned objects is scanned across the scanfield image and serves as input for the neural network. The neural network attempts to match the windowed image with one of the images it has learned.
- 183 – **equake**: The program simulates the propagation of elastic waves in large, highly heterogeneous valleys (i.e. California's San FernandoValley). The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method.
- 187 – **facerec**: This is an implementation of the face recognition system described in M.Lades et al. (1993), IEEE Trans. Comp. 42(3):300-311. An object is a set of faces photographed frontally and represented as labeled graphs. The graph is a regular grid. To each vertex of the grid graph a set of features are attached; they are computed from the Gabor wavelet transform of the image and represent them in the surroundings of a vertex. An edge of the graph is labeled with the vector connecting its two vertices and represents the topographical relationship of those vertices. An object represented in this way is compared to a new image in a process called elastic graph matching. This is done by first determining the Gabor wavelet transform for the new

image. Then, for a given correspondance between the graph's vertices and a set of image points, a function taking into account both the similarity of the feature vectors at every vertex and its corresponding image point, and the distortion of the graph generated by the set of image points, measured as the change in the edge labels, can be computed. This graph similarity function is then the objective function of an optimization process that varies the set of corresponding points in the image. This optimization process is implemented in two steps: The global move step keeps the graph rigid and moves it systematically over all of the image, resulting in a placement that has the highest similarity to the graph. This step can be considered as finding the object (face) in the image. The local move step then takes this placement as the starting position, and visits every vertex in random order. At each vertex, the similarity function is evaluated on a small subgrid surrounding the current position. (This is a small change from the algorithm as originally published, where the trial moves at each node were random as well.) If the similarity function's value is improved at one of those positions, the change is made permanent; such a move is called a hop. One round visiting each vertex position is called a sweep. The local move step terminates when a sweep is completed without a hop having been performed. The benchmark consists of the following main phases:

1. Face Learning: The system has no prior knowledge of the class of object it is supposed to recognize. It "learns" this by extracting a canonic graph from one so-called canonic image; that image and the position at which the graph is to be extracted are specified by the user.
2. Graph Generation: For each of the images in the album gallery, the Gabor wavelet transform is computed, and the global move step is performed using the canonic graph. The resultant graph is extracted from the transform and stored.
3. Recognition: For each of the images in the probe gallery, the Gabor wavelet transform is computed, and the global move step is performed using the canonic graph. Then, a local move step is performed using each of the stored graphs. The resultant vector of similarity values is searched for the maximal value; the associated graph (and image) indicates the object recognized.

The parts that take the most computational time have the following characteristics:

1. Gabor Wavelet Transform is performed by computing the forward fast Fourier transform (FFT) of the image, multiplying it with a number (here, 40) of kernels, computing the backward FFT for each of the results, and inserting the absolute value for each pixel into a two-dimensional array of feature vectors
2. A Global Move is dominated by the computation of the feature similarity function, which basically is the scalar product of the two feature vectors.
3. A Local Move is dominated by the computation of the similarity function. The distortion of the grid introduced by the hops has to be taken into account. This is done incrementally, i.e., the contribution of the vertex currently being considered to the distortion is computed for both its old and its new position, which entails handling the different positions a vertex can be in the graph. During each recognition step, practically all of the code is exercised.

- 188 – **ammp**: it runs molecular dynamics (i.e. solves the Ordinary Differential Equation defined by Newton's equations for the motions of the atoms in the system) on a protein-inhibitor complex which is embedded in water (see Harrison 1993 for de-

scriptions of the algorithm and stability analysis on it). The energy is approximated by a classical potential or "force field". The protein is HIV protease complexed with the inhibitor *indinavir*. There are 9582 atoms in the water and protein making this representative of a typical large simulation. This benchmark is derived from published work on understanding drug resistance in HIV (Weber and Harrison 1999).

- 189 – **lucas[not collected]**: it performs the Lucas-Lehmer test to check primality of Mersenne numbers 2^p-1, using arbitrary-precision (array-integer) arithmetic. Accomplishes the Mersenne-mod squaring via the discrete weighted transform technique of Crandall and Fagin (Math. Comp. 62 (205), pp.305-324, January 1994). Uses a data-local, cache-friendly FFT to efficiently perform the large-integer squaring of the Lucas-Lehmer iterations.
- 191 - **fma3d**: it is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads. As an explicit code, the program is appropriate for problems where high rate dynamics or stress wave propagation effects are important. In contrast to programs using implicit time integration algorithms, the program uses a large number of relatively small time steps, with the solution for the next configuration of the body being explicit (and inexpensive) at each step. To further reduce the computational effort, the program has a complete implementation of Courant subcycling in which each element is integrated with the maximum time step permitted by local stability criteria. For simulations that have large differences in element critical time steps over the mesh, very significant savings in execution time are achieved. There are no inherent limits on the size of an analysis model, and storage allocation is dynamic within the code. The program may be applied to static and quasi-static problems either by using the dynamic relaxation option or by simply applying the external loads slowly and integrating the dynamics equations until all significant transients have died out. The algorithms and architecture of the program are designed for accuracy and robustness. The solution portion of the program is in a form suitable for cache-based computer hardware architectures.
- 200 – **sixtrack**:The function of the program is to track a variable number of particles for a variable number of turns round a model of a particle accelerator such as the Large Hadron Collider (LHC) to check the Dynamic Aperture (DA) i.e. the long term stability of the beam.

CINT2000

- 164 – **gzip**:(GNU zip) is a popular data compression program written by Jean-Loup Gailly using Lempel-Ziv coding. It performs no file I/O other than reading the input. All compression and decompression happens entirely in memory
- 175 – **vpr**: it is a placement and routing program; it automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. VPR is an example of an integrated circuit computer-aided design program, and algorithmically it belongs to the combinatorial optimization class of programs. Placement consists of determining which logic block and which I/O pad within the FPGA should implement each of the functions required by the circuit. The

goal is to place pieces of logic connected (i.e. must communicate) close together in order to minimize the amount of wiring required and to maximize the circuit speed. VPR uses simulated annealing to place the circuit. An initial random placement is repeatedly modified through local perturbations in order to increase the quality of the placement. Routing (in an FPGA) consists of determining which programmable switches should be turned on in order to connect the pre-fabricated wires in the FPGA to the logic block inputs and outputs, and to other wires, such that all the connections required by the circuit are completed and such that the circuit speed is maximized. The connections required by the circuit are represented as a hypergraph, and the possible connections of wire segments to other wires and to logic block inputs and outputs are represented by (a different) directed graph, which is often called a "routing-resource" graph. VPR uses a variation of Dijkstra's algorithm in its innermost routing loop in order to connect the terminals of a net (signal) together. Congestion detection and avoidance features run "on top" of this innermost algorithm to resolve contention between different circuit signals over the limited interconnect resources in the FPGA.

- 176 – **gcc**: it is based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled. It has had its inlining heuristics altered slightly, so as to inline more code than would be. This was done so that it would spend more time analyzing it's source code inputs, and use more memory. Without this effect, it would have done less analysis, and needed more input workloads to achieve the run times requirements.
- 181 – **mcf**: it is designed for the solution of single-depot vehicle scheduling problems occurring in the planning process of public transportation companies. It considers one single depot and a homogeneous vehicle fleet. Based on a line plan and service frequencies with fixed departure/arrival locations and times are derived. Each of these timetabled trips has to be serviced by exactly one vehicle. These trips are called *dead-head* trips. In addition, there are *pull-out* and *pull-in* trips for leaving and entering the depot. Cost coefficients are given for all trips. It is the task to schedule all timetabled trips such that the number of necessary vehicles is as small as possible and, subordinate, the operational costs among all minimal fleet solutions are minimized. For the considered single-depot case, the problem can be formulated as a large-scale minimum-cost flow problem that we solve with a network simplex algorithm accelerated with a column generation. The network simplex code "MCF Version 1.2” is the core of the implementation. The network simplex algorithm is a specialized version of the well known simplex algorithm for network flow problems.
- 186 – **crafty**: it is a high-performance Computer Chess program that is designed around a 64bit word. It can be configured to run a reproducible set of searches to compare the integer/branch prediction/pipe-lining facilities of a processor.
- 197 – **parser**: it does the grungy job of chopping the user's input sentence into words, processing the special commands, and calling all the functions necessary to parse the input sentence
- 252 – **eon**: it is a probabilistic ray. It sends a number of 3D lines (rays) into a 3D polygonal model. Intersections between the lines and the polygons are computed, and new lines are generated to compute light incident at these intersection points. The final result of the computation is an image as seen by camera. The computational de-

mands of the program are much like a traditional deterministic ray tracer as described in basic computer graphics texts, but it has less memory coherence because many of the random rays generated in the same part of the code traverse very different parts of 3D space.

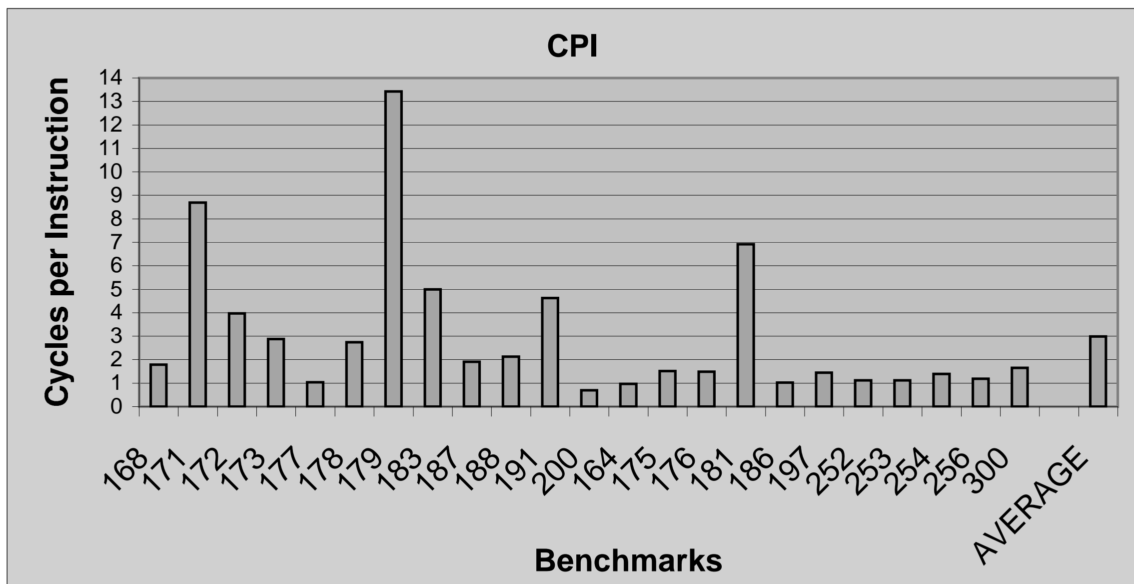
- 253 – **perlbmk**: it is a cut-down version of Perl v5.005_03, the popular scripting language. It has had most of OS-specific features removed. In addition to the core of Perl interpreter, several third-party modules are used.
- 254 – **gap**: It implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).
- 255 – **vortex [not collected]**: it is a single-user object-oriented database transaction benchmark. It is a derivative of a full OODBMS that has been customized to conform to SPEC CINT2000 (component measurement) guidelines
- 256 - **bzip2**: it is based on Julian Seward's bzip2 version 0.1. It does not perform file I/O other than reading the input. All compression and decompression happens entirely in memory.
- 300 – **twolf**: it stands for TimberWolfSC and it determines the placement and global connections for groups of transistors (known as standard cells) that constitute the microchip. The placement problem is a permutation problem and the program uses simulated annealing as a heuristic to find very good solutions for the row-based standard cell design style. The basic simulated annealing algorithm has not changed since its inception in 1983. This version of TimberWolfSC has been customized so that it would have a behavior that captures the flavor of many implementations of simulated annealing.

Experimental Results

The experimental results collect the 32 Events for each of the SPEC2000 benchmarks; from them, we can infer characteristics of the microprocessor. We focus on the average behavior of the architecture more than application-based. The set of applications is suited for modern microprocessor architectures and, when useful for the clarity of he explanation, considerations on the single benchmark are performed. The interpretation of experimental results can be summarized as follows: **CPI, Data Miss Ratio, Instruction Miss Ratio, Secondary Cache way Prediction, Pre-fetch, Branch Prediction** and the common **FLOPS** and **INTPS.**
The charts presenting the experimental results offer a column-based picture of the results per benchmark and the last column is the average behavior. The *floating point* benchmarks are the first twelve and the *integer* benchmarks are the last eleven.

CPI, Cycles per Instruction

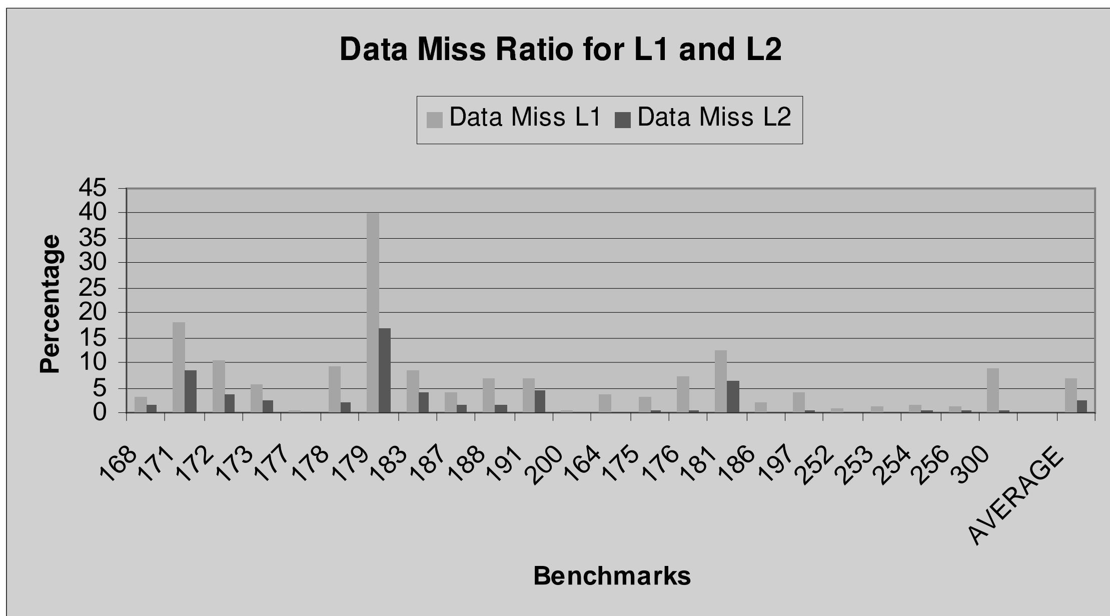
R12K can fetch multiple instructions per cycle, it can issue multiple instructions per cycle and it can execute multiple instructions per cycle. The number of cycles over the number of graduated instruction represents the slow down with respect to pure sequential 1-cycle-latency processor when greater than one; otherwise it represents the speed up with respect to sequential 1-cycle-latency processor. It is a qualitative measure and it must be interpreted carefully.



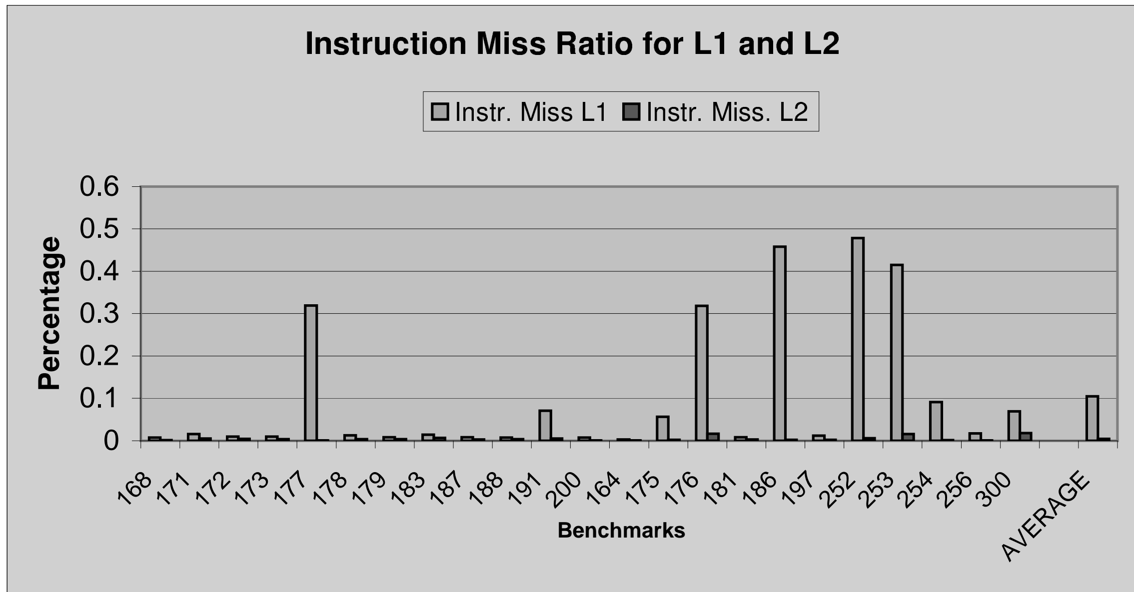
At the first glance, the R12K performance is very poor achieving in average 3 cycles per instruction. Most of the time the problem it is not due to bad design but data starvation of the units.

Miss Ratio

The microprocessor can stall either when not enough instructions are fetched from the instruction cache or not enough data can be read from the data cache. Cache miss is the subject of the following charts.



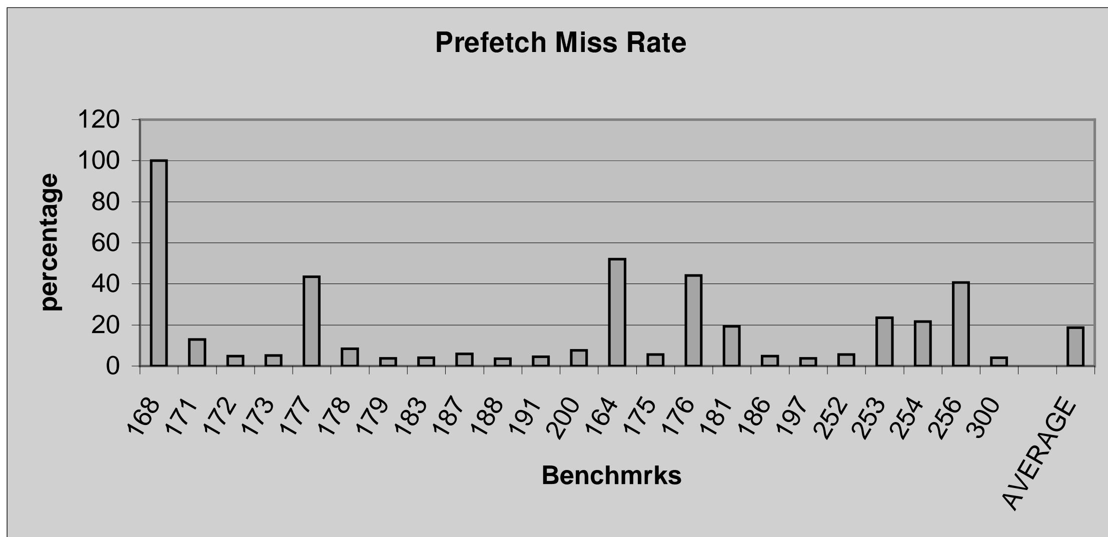
In this chart we can see the data miss ratio for the first level of cache and for the secondary cache. We can notice that the application with the larger CPI has the worst miss ratio: 40% and 15% respectively for L1 and L2. The slow down is mainly due to data starvation. But the average data miss ratio is 7% for the first level of cache and 2.5% for the second level, which is pretty good.



The instruction miss ratio is very small, no more than 0.5% and in average is 0.1%.
If the processor stall for starvation is because of data. Another investigation has proven that the misses are mostly capacity misses and therefore the only way to improve the data cache performance is by increasing the cache size.

Pre-fetch

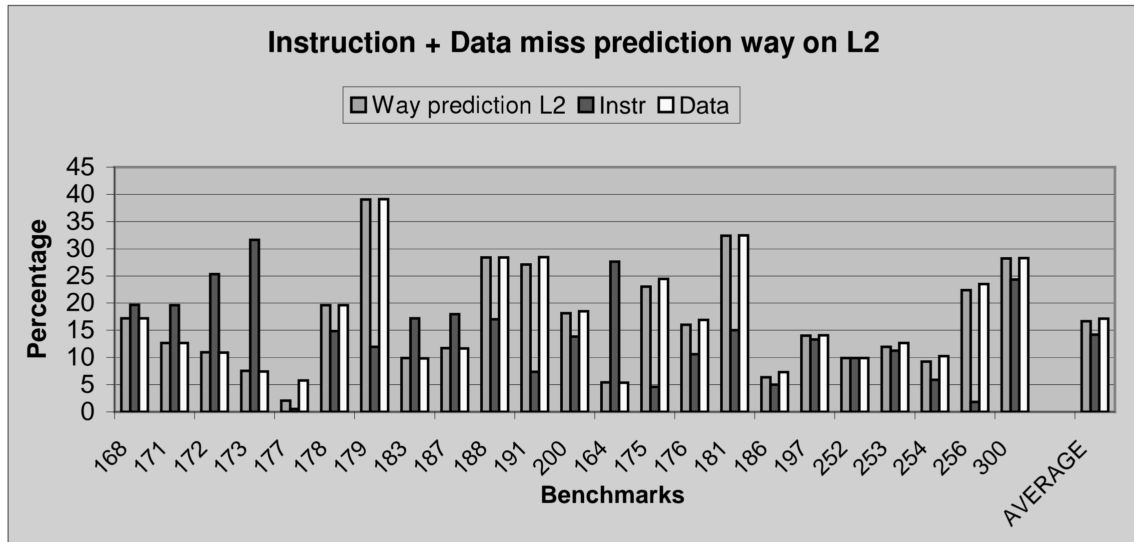
Data pre-fetching can be a solution, we can measure the pre-fetch miss rate.



When a pre-fetch instruction does access a line not available, it is useful, otherwise is redundant. Therefore a high ratio is desirable. In fact, the percentage is in average 15%. Most of the pre-fetches are redundant. Benchmark 168 achieve a 100% miss ratio because it has not pre-fetches at all.

Secondary Cache Way Prediction

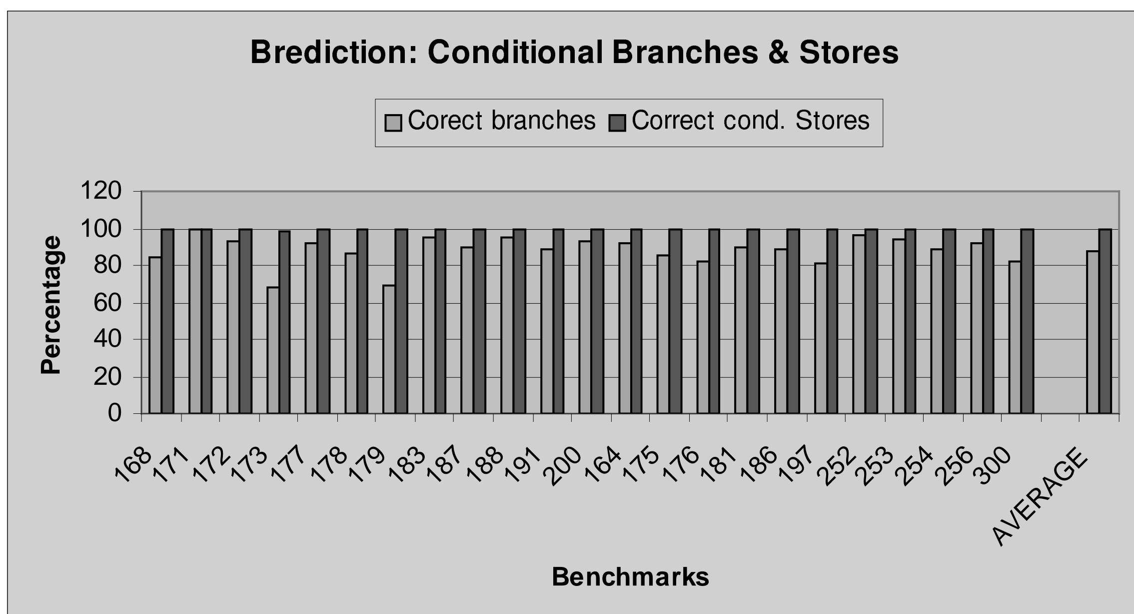
The secondary cache is 2-way associative, but the search of a data is not done in parallel for each block. Indeed, the search is done sequentially and the choice of the first array is guessed.



In this chart we introduce the percentage of incorrectly predicted data fetch, incorrectly predicted instruction fetch and both. In average the way is incorrectly predicted a little bit more than 15%. Which is not very low. Note that for this particular measure a small miss prediction percentage is a good percentage.

Branch Prediction

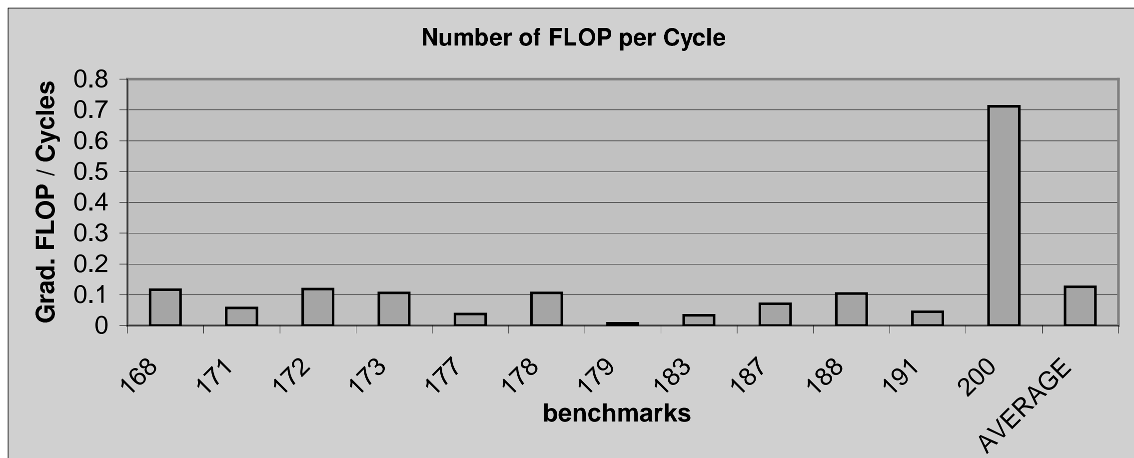
The complex logic to guess correctly the flow of a program is advantageous if the percentage of correctly predicted branch is high (the higher the better), because it thwarts the chaotic instruction stream due to conditional branches, ideally having a single block of instructions easy to fetch.



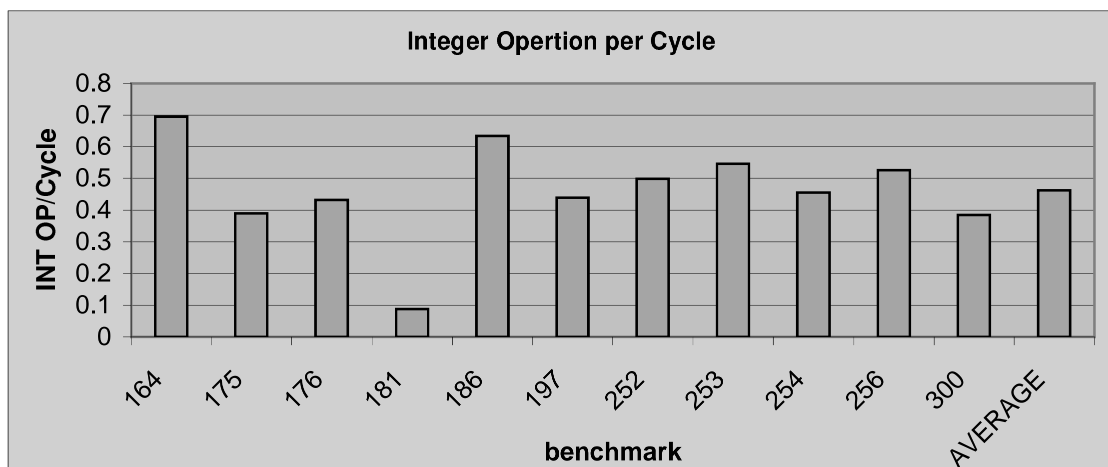
In average the prediction is correct 88% of the times. For the size of the array used to store the history of the taken branches, the percentage obtained is not high as expected. In previous papers we can find percentage around 95%. Observe that the lowest percentage is achieved for *scientific programs*. For which it is common assumption that conditional branches are mostly loop bounds checking and therefore easy to predict (relatively small inner loops can become a big problem for a branch predictor with short memory). Otherwise it may be fault of the compiler, which it may be not able to substitute conditional branches with conditional moves and stores. From the same chart we can see that the percentage of conditional stores is very high. In fact, there are a very few conditional store instructions respect to branch instructions and they have no contribution.

FLOPS-INOPS?

A very common measure of performance is the FLOPS (floating point operations per second), we introduce the measure floating point instructions per cycle, which is independent from the length of cycle.



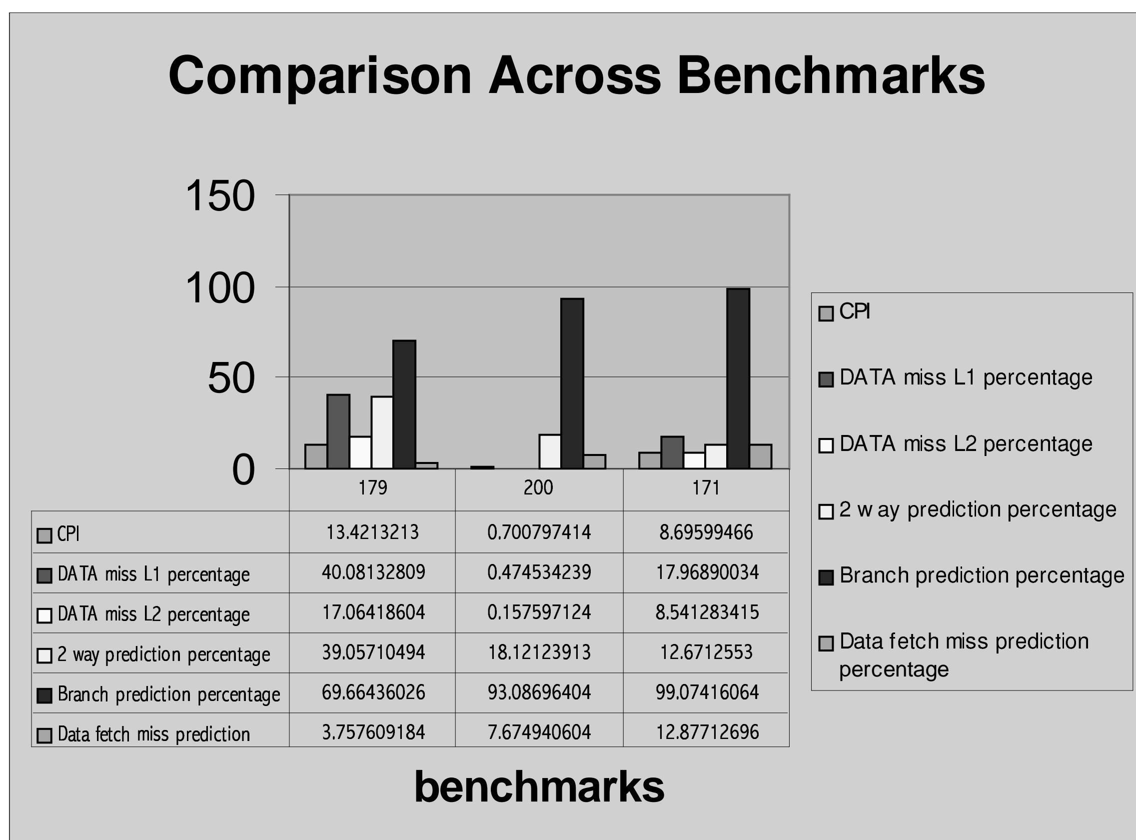
We can notice this architecture can issue from one to two floating point instructions per cycle, but in general the floating point units are under utilized.
We can determine how much the ALUs are utilized in the same fashion as introduced in the following
chart



The average number of integer instructions is one every other cycle. Note that this is a lower estimation since some instructions occupy two entries and therefore they are counted twice.

Comparison Across Benchmarks

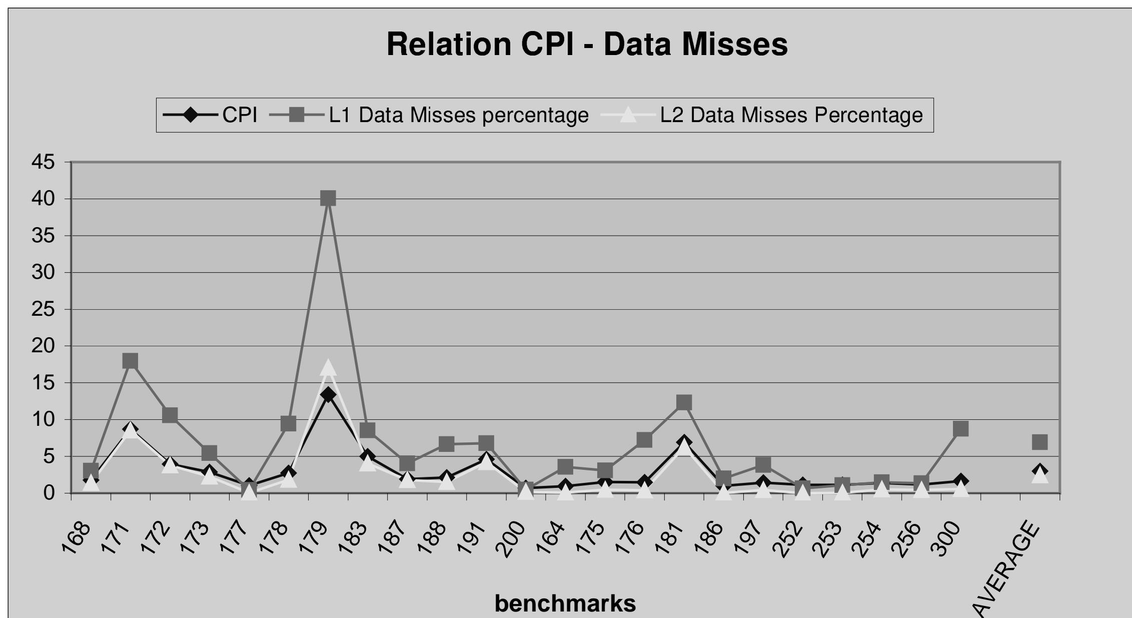
In this section we are going to investigate in more detail two benchmarks, the one with the best CPI and the one with the worst CPI. A third one is used as benchmark with intermediate performance. This section is used to offer a details interpretation of why performance is achieved. The following chart is used to stress out the relationship of the events measured:



	179	200	171
CPI	13.4213213	0.700797414	8.69599466
DATA miss L1 percentage	40.08132809	0.474534239	17.96890034
DATA miss L2 percentage	17.06418604	0.157597124	8.541283415
2 way prediction percentage	39.05710494	18.12123913	12.6712553
Branch prediction percentage	69.66436026	93.08696404	99.07416064
Data fetch miss prediction	3.757609184	7.674940604	12.87712696

Benchmark 179 has the larger CPI (13), benchmark 200 has the smallest (0.7) and 171 is in the middle. Benchmark 200 has very small data miss ratio and very often the branch prediction logic guessed correctly. Every cycle the processor executes two instructions, and every four instruction one is a floating point instruction. Benchmark 200 experimental results show how the processor is not fully used yet. An application with very low mss ratio and with high percentage of correct branch prediction, it should have better performance, unless the scheduling of instructions and register allocation is not "*perfect*". The dynamic scheduling is based on a very short list of instructions; the SGI compiler may not perform a good instruction scheduling and register allocation. A non optimal instruction scheduling can introduce bubbles into the pipes. Non optimal register allocation can introduce spilling to/from the memory; the spilling due to register allocation may introduce memory traffic with very high hit percentage but also bubbles into the pipes

because the value required must be read from the cache instead from register. Spilling it may be one of the reason the benchmarks obtain low in average data miss ratio, since it increases artificially the number of memory accesses but not increasing very much the misses. If the best executable cannot utilize fully the processor, none can do it. The other applications may amplify the negative effects so introduced. Two possible ways to check this consideration is either to use a different compiler and produce a different executable or investigate the assembly code of the application.
In general we can see that 50% of data misses at the first level L1 are misses for the secondary cache. The caches are not blocking but the processor can handle a finite number of misses at anytime (the MHT has just 5 entries and four of them are for internal misses). Therefore when an access is very often a hit only in memory, the high latency let the miss hang into the MHT for long time and increase the chance to stall the processor. The experimental results suggest that high CPI is associated with high data miss.
In the following chart it is qualitatively shown the "*correspondence*" between them.



Of course, they differ not by a constant. There is not a linear relationship between stall time, latency misses and number of misses, because latency can be partially hidden in different ways. Nonetheless the shape of peaks and valleys show the tight relationship between these two measures (in particular the L1 data miss ratio is the amplified wave form of the CPI). With the premises there are mostly capacity misses, and the data pre-fetching logic does not work, the architecture cannot avoid most of the misses; it could hide them through a different instruction scheduling and software pre-fetching (compiler stuff).

Details: optimization flags and execution

The executables has been generated by SGI compiler **cc** with the default optimization flag **–Ofast**. The default optimizations are for R10K microprocessor. We could not produce an executable optimized for the host architecture (R12K IP32). From personal experience, the choice of optimization flags and compiler is very important. Therefore it

may be that some of the loss of performance is due to the compiler. SPEC2000 benchmark suit comes with a predefined shell environment, where the final user can tune up the compiler flags based on pre-defined set of inputs. For the publication of the performance measurements the *reference* set must be used. For example to collect performance hardware counter statistics, in particular for benchmark 171, we type the following command:

```
perfex –a  runspec --config=sgi.2.cfg --tune=base --size=ref --
iterations=5 --action run 171
```

The script "`runspec`" executes the benchmark but it does not produce any output file as documentation, and it does not check for the correctness of results. The input is the reference input. We verified the performance measure running the benchmarks as single application from console and the experimental results so obtained differ by a relative error of 1% (which is comfortably small).

Conclusions

We would suggest reading the experimental results considering the chance that the compiler does not exploit fully the microprocessor architecture. FLOPS (INOPS) is the first measure where the microprocessor partially fails, i.e. one floating point every ten cycles, in average, is a very poor performance. Another compiler may provide better executables for some of the benchmarks, and worse form some others. We did not compare the performance of a different compiler but a further investigation in this direction is beyond the scope of this report. We can summarize our work in a few considerations.

- Branch prediction works effectively (88% of the branches are guessed correctly) but not as expected (~95%). In practice one branch every six is predicted incorrectly and the instruction speculation must be aborted and the execution must roll back
- The memory hierarchy seems the main bottleneck for performance even if the average miss ratio is modest in size (7% data miss ratio and 2% instruction miss ratio in average).
- The secondary cache way prediction logic is not very effective.
- Data cache pre-fetching is not effective.

The architecture seems not fully utilized, the parallel units most of the times are idle, it may be waiting for data.

The performance hardware counters [3] are useful tools for performance evaluation of microprocessors. It is a non-invasive tool to measure microprocessor performance and utilization, easy to use and, one the microprocessor architecture is known, easy to interpret.

References:

[1] MIPS R10000 Microprocessor User Manual Version 2

[2] An Illustration of the Benefits of the MIPS R12000 Microprocessor and OCTANE System Architecture. http://www.sgi.com/octane/images/octane.pdf

[3] Marco Zagha, Brond Larson, Steve Turner and Marty Itzkowitz "Performance Analysis Using the MIPS R10000 Performance Counters' *Proc. Supercomputing 1996, Nov. 96, Pittsburgh, PA.* http://www.sgi.com/processors/r10k/timing/perf_count.html

[4] SPEC CPU 2000, benchmark documentation CFP2000 http://www.spec.org/osg/cpu2000/CFP2000 CINT2000 http://www.spec.org/osg/cpu2000/CINT2000