# Energy saving through a simple load control mechanism

Tanausú Ramírez
DAC - UPC
D6-113 Campus Nord
Barcelona, Spain
tramirez@ac.upc.edu

Alex Pajuelo
DAC - UPC
C6-205 Campus Nord
Barcelona, Spain
mpajuelo@ac.upc.edu

Oliverio J. Santana
DIS - ULPGC
s5 - Campus Tafira
Las Palmas de GC, Spain
ojsantana@dis.ulpgc.es

Mateo Valero[*]
DAC - UPC
D6-201 Campus Nord
Barcelona, Spain
mateo@ac.upc.edu

## ABSTRACT

To alleviate the memory wall problem, current architectural trends suggest implementing large instruction windows able to maintain a high number of in-flight instructions. However, the benefits achieved by these recent proposals may be limited because more instructions are executed down the wrong path of a mispredicted branch. The larger number of misspeculated instructions involves increasing the energy consumed compared to traditional designs with smaller instruction windows. Our analysis shows that, for some SPEC2000 integer benchmarks, up to 2,5X wrong-path load instructions are executed when the instruction window of a 4-way superscalar processor is increased from 256 to 1024 entries.

This paper describes a simple speculative control technique to prevent wrong-path load instructions from being executed. Our technique extends the functionality of the load-store queue to block those load instructions that depend on a hard-to-predict conditional branch until it is resolved. If the branch is actually mispredicted, unnecessary cache misses can be avoided, saving energy down the wrong path. Furthermore, instructions that depend on a blocked load are not issued because their source values are not available, which also saves dynamic energy. Our results show that the proposed mechanism reduces, on average, up to 26% misspeculated load instructions and 18% wrong-path instructions without any performance loss.

## Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Design Styles: Cache memories*
; B.8 [**Hardware**]: Performance and Reliability—*General, Performance Analysis and Design Aids*; C.1.3 [**Processor Architecures**]: Other Architecture Styles—*Pipeline processors, out-of-order processors*

[*]Professor Mateo Valero is also member of the Barcelona Supercomputing Center (BSC - CNS) email: mateo@bsc.es.

## General Terms

Measurement, Performance, Experimentation

## Keywords

Kilo-instruction processors, Branch Prediction, Confidence Estimation, Energy Saving

## 1. INTRODUCTION

Branch prediction is a common and effective speculative technique faced to alleviate the problems arising from control dependences. Branch predictors allow predicting the outcome of branch instructions, enabling the processor to exploit the available instruction level parallelism (ILP) before branch resolution. However, when a branch is mispredicted, the processor must recover from the execution of wrong path instructions, discarding all the speculative work done.

Despite current processors employ aggressive branch prediction techniques with high branch prediction accuracy (over 97% on average), processors still continue fetching and executing useless speculative instructions down the mispredicted paths. This phenomenon leads to lost performance opportunities and wasted energy due to the execution of useless misspeculated instructions.

Moreover, branch mispredictions generate wrong-path speculative memory references that may be harmful to processor performance. A misspeculated memory reference could fetch data into the cache hierarchy that will not be used by correct-path execution, and thus it could pollute the caches by eliminating useful cache blocks. Even if a branch is mispredicted and the wrong path execution goes through a memory-intensive section, it may cause resource and bus conflicts, delaying correct-path memory accesses and thus hindering overall performance.

This problem becomes worse in novel designs focused on alleviating the memory wall problem. Several authors [1, 5, 6, 19] propose to virtually enlarge the instruction window, increasing the amount of independent instructions available for execution while long-latency loads are outstanding in memory. Thus, while a memory access is being resolved, the processor is able to overlap it with the execution of thousands of instructions.

Accurate branch prediction is more important in these scenarios, since processors are not limited by the number of entries in the instruction window. For example, if a mispredicted branch depends on a long-latency load, the amount of wrong-path instructions executed is higher than in traditional designs with smaller instruction windows, since it is less likely that the fetch and decode stages stall due to the lack of entries in the instruction window. Moreover, cache pollution is more probable because more memory accesses can be performed down the wrong path of a mispredicted branch. All these issues point out large instruction window processors will suffer from higher energy consumption due to wrong-path execution.

In this paper, we propose a simple technique to reduce the energy consumption of wrong path execution in large instruction window processors. The mechanism uses a confidence estimator to detect branches that are likely to be mispredicted by the branch predictor. As soon as a branch is estimated as a low confidence branch, the processor blocks all subsequent load instructions, preventing them from accessing the first level data cache and the higher levels of the memory hierarchy. Moreover, the instructions that depend on blocked loads are also blocked, since not all their source values can be computed. We have developed a special confidence estimator (Decoupled Counters) that has better behavior than similar estimators for our Speculative Load Control Mechanism.

Once the branch is resolved, blocked loads are allowed to access the cache hierarchy if the prediction was correct. Otherwise, if the branch was mispredicted, blocked loads are squashed, saving memory accesses and energy. Our results show that a simple technique applied to the load-store issue queue prevents more than 26% of misspeculated loads from accessing the cache, decreasing energy consumption. Moreover, we will show that this energy reduction does not lead to any performance degradation.

The reminder of this paper is organized as follows. The motivation for the mechanism is presented in Section 2. Section 3 describes the design and implementation issues of our proposal. Section 4 describes our experimental framework. Section 5 presents the evaluation results for the mechanism. Section 6 reviews related work on analyzing the performance impact and energy savings of wrong-path execution. Finally, Section 7 concludes the paper.

## 2. IMPACT OF WRONG-PATH INSTRUCTIONS

Current design trends rely on higher amounts of speculative execution to improve processor performance. However, completely avoiding misspeculations is not possible. In addition, increasing the number of speculative instructions executed per prediction involves increasing the number of wrong-path instructions, and thus there is a higher waste of energy due to instructions that are executed but not actually committed.

The energy consumed by a processor is related to the total amount of instructions executed to complete a program. Figure 1 shows the amount of misspeculated instructions for two processors with different reorder buffer sizes. Framework configuration and benchmark simulation details can be found in Section 4. The larger the instruction window, the more instructions can be executed down the wrong-path of a mispredicted branch. In this particular case, on average, 66% more instructions are executed when the reorder buffer is enlarged from 256 (light bars) to 1024 (dark bars) entries, being *vpr* and *twolf* the benchmarks that exhibit the larger increment of misspeculated instructions (2,5X in both cases).

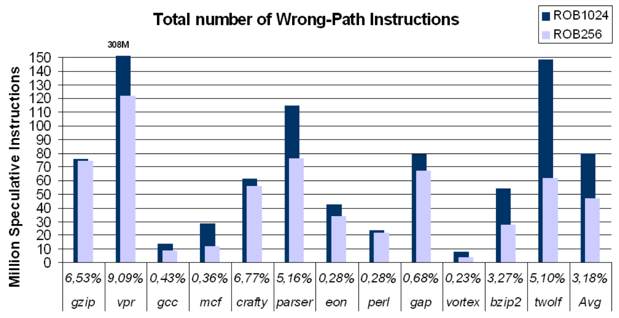

**Figure 1: Total instructions executed down the wrong-path (missprediction rate is showed in X-axis)**

Since more misspeculated instructions are executed, the total number of wrong-path memory references is also increased, as shown in Figure 2. Light bars correspond to a processor with a 256-entry reorder buffer and dark bars show results obtained with a larger reorder buffer (1024 entries). On average, the 1024-entry instruction window processor executes 1,6X more L1 data cache accesses down the wrong-path compared to the 256-entry instruction window processor. The major drawbacks of these extra speculative loads are cache pollution as well as bus contention, which may harm the performance of correct-path load instructions.

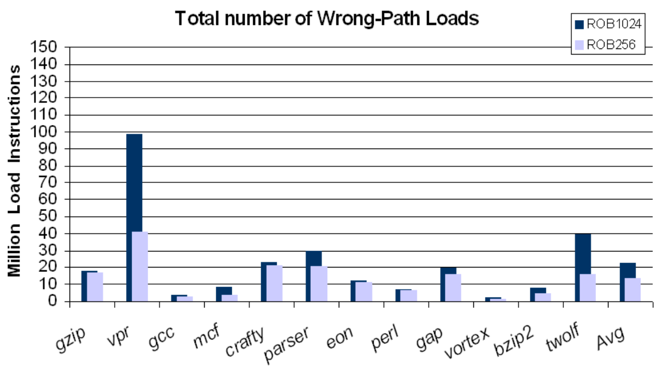

**Figure 2: Total Loads executed down the wrong-path**

We follow a simple approach to measure the performance impact rate of wrong-path load instructions. We use an oracle predictor to transform misspeculated loads to *NOP* instructions at the issue stage, avoiding data cache accesses down the wrong path, and thus any possible impact of them on processor performance. That is, either the possible beneficial effects due to wrong-path prefetching or detrimental effects caused by cache pollution are avoided.

Figure 3 shows IPC variation of a 1024-entry instruction window processor that does not send wrong-path memory requests to the data cache; the baseline is the same 1024-entry instruction window processor without this restriction. Most benchmarks achieve little performance variation (1,27% improvement on average) when misspeculated loads are not issued to the memory system and cache pollution is avoided. The best case is the *vpr* benchmark, which achieves 9,9% IPC improvement.

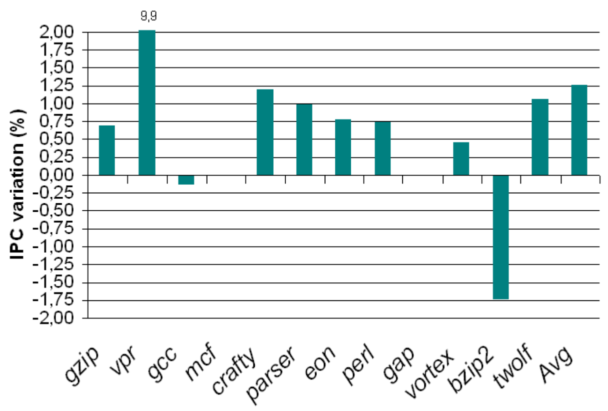

**Figure 3: IPC variation of a 1024-entry instruction window processor when wrong-path instructions are not allowed to access memory**

But not all misspeculated loads are prejudicial. Some wrong-path memory accesses perform useful prefetch, bringing data to the memory hierarchy that will be accessed later by correct-path load instructions [12]. For instance, removing this prefetch effect is what makes that two benchmarks, *bzip2* and *gcc*, present little performance degradation in Figure 3 (-1,74% and -0,14% respectively). In fact, most of this prefetch down the wrong path is performed by control-independent loads. Those are loads that are executed down all paths of a conditional branch. Section 3.3 describes a simple mechanism to detect those loads. The performance effect of these loads are later discussed in Section 5.

## 3. SPECULATIVE LOAD CONTROL MECHANISM

In general, large instruction window processors execute a high amount of wrong-path speculative instructions, including speculative loads. These loads consume extra energy and, as previously shown, they may have a negative impact on performance. Based on this observation, we propose a simple speculation control mechanism to reduce energy usage down wrong speculative paths, also avoiding performance degradation due to cache pollution and bus contention effects.

Our mechanism works in two steps. First of all, as soon as a branch instruction is predicted, a confidence estimator [8][7] determines how likely the branch prediction is correct. In this sense, the confidence estimator divides branches into *high confidence branches* (a branch is likely to be correctly predicted) and *low confidence branches* (there is a high probability of a branch misprediction). The association of the branch predictor and the confidence estimator can produce four possible combinations:

- Correct prediction and high confidence ($C_{HC}$): The branch is correctly predicted by the predictor and the estimator agrees.
- Correct prediction and low confidence ($C_{LC}$): The branch is correctly predicted but the estimator estates that this prediction is wrong (a.k.a. *false positive*).
- Wrong prediction and high confidence ($W_{HC}$): In this case the estimator does not detect that the prediction is wrong.
- Wrong prediction and low confidence ($W_{LC}$): The branch is mispredicted and the estimator detects this situation.

The second step of the mechanism effectively blocks load instructions. Once a branch is estimated to be low confidence, the processor enters in the *Low Confidence Estimation State* ($LCES$), and marks this situation by setting a new 1-bit register named *Low Confidence Estimation Register* ($LCER$) to 1. In this state, the processor blocks the loads following the low confidence branch until the prediction for the latter is checked. Once the branch prediction is resolved, the processor leaves the $LCES$ state, setting $LCER$ to 0 and either unblocking the loads if the prediction is correct or squashing them otherwise.

Accurate confidence estimation is fundamental for our technique to perform adequately. The optimal situation is classifying all mispredicted branches as low confidence and all correctly predicted branches as high confidence. Those mispredicted branches that are classified as high confidence ($W_{HC}$) are lost opportunities to apply our mechanism, but they do not involve losing performance. On the contrary, accurately identifying mispredicted branches is not enough: those correctly predicted branches identified as low confidence ($C_{LC}$) will involve blocking correct-path loads unnecessarily, delaying the memory accesses and thus, degrading processor performance.

### 3.1 Confidence Estimation

Since our proposal works over *low confidence* branches, we need a confidence estimator to determine if a predicted branch is likely to be mispredicted. According to the four confidence estimator metrics cited in [7] to characterize the goodness of confidence estimators, we focus in two of them for mispredicted branches: *specificity* (SPEC) and *predictive value of a negative test* (PVN). SPEC represents the fraction of incorrect branch predictions identified by the confidence estimator as low confidence. PVN represents the probability that a low confidence estimation is correct. Good quality confidence estimators focus on our approach ideally should have high values from both SPEC and PVN. However, as these factors must be balanced, increasing SPEC usually means decreasing PVN and vice versa.

Mainly, a high PVN is desirable, that is, we want to ensure, as far as possible, that a low confidence branch is actually a mispredicted branch, since blocking correct-path loads would severely degrade processor performance. The drawback is that increasing PVN reduces SPEC, making the confidence estimator more conservative because less mispredicted branches will be considered as low confidence. However, losing these opportunities is less prejudicial for performance than considering a correct predicted branch as low confidence, since the mechanism would block correct-path loads delaying their memory accesses.

There are several confidence estimators presented in the literature [17][8]. Our first implementations used the Saturating Counters and Resetting Counters confidence estimators described in [8]. However, although we reduce the total number of speculative load instructions, we also get an average performance loss around 3% because of correctly-predicted branches classified as low confidence. To achieve better performance results, we modified the estimator implementation by decoupling the saturating counters into two separate counters. One of these counters is used for branches that are predicted taken and the other one is used for branches that are predicted not taken.

Since most branches are biased toward taken or not taken direction, confidence counters usually use just one of the counters to estimate the behavior of the branch. As a consequence, a taken branch and a not taken branch that coincide in the same table entry will not compete for the same counter, reducing aliasing and thus improving confidence accuracy. Hard-to-predict branches will likely use both counters due to their seemingly random behavior, but having two counters devoted to the branch will also help in improving the accuracy of confidence predictions for this kind of branches, since they make it possible to hold more information about the branch.

Finally, once counters are decoupled, they must be sharply tuned up to fit the mechanism requirements for our purpose, that is, reducing as many misspeculated cache accesses as possible without any performance degradation. The most significant parameters of the estimator setup are: *table size, bits per counter, initialization value, increment update factor*, and *decrement update factor*. We chose the best configuration that presents higher $PVN$ without loosing too much $SPEC$. In particular, the setup selected for the Decoupled Counters confidence estimator uses a 16K-entry table with two 4-bit saturated counters for both taken and not taken branches. The counters are initially set to 10 and this value is modified when the estimated branch is finally committed. When the branch predictor correctly predicts the branch outcome, the corresponding counter (taken or not taken) is increased by *2*. If the branch prediction was wrong, the counter is decreased by *1*, that is, correct predictions have higher weight than mispredictions. Branches are considered high confidence if the corresponding counter has a value equal or greater than the confidence threshold. In our case, the optimal setup considers a branch prediction as low confidence if the counter value is below 8, that is, if the most representative bit is not set.

| **Confidence Estimator** | **SPEC** | **PVN** |
|---|---|---|
| *Saturating Counters* | 65% | 31% |
| *Reseting Counters* | 73% | 29% |
| *Decoupled Counters* | 37% | 87% |

**Table 1: Confidence Estimators Metrics**

Table 1 shows the $SPEC$ and $PVN$ values obtained for the confidence estimator metrics using three different confidence estimators: the Saturating Counters scheme [17], the Reseting Counters scheme (JRS) [8], and our Decoupled Counters scheme. Although [10] suggests that it is better to use a confidence counter with $SPEC$ higher than $PVN$, our performance results show that, due to the aforementioned reasons, the best choice for our proposal is the Decoupled Counters configuration that presents higher $PVN$ than $SPEC$.

## 3.2 Speculative Load Blockade Control

One important design point in the mechanism is how and where to block the low confidence loads. In fact, since the main objective of the mechanism is to avoid misspeculated memory accesses, any structure involved in holding the pending load instructions can be selected to perform the blockade.

In order to implement a general approach, we decided to use the load-store issue queue (LSIQ) to block the speculative load instructions. To mark those loads that depend on a low confidence branch, only one bit per entry is needed, namely the *Confidence Branch Estimate bit* ($CBE$ bit). This bit is used to set each load as low confidence (0) or high confidence (1) depending on the estimation of the last predicted branch. The value of this bit is obtained by inverting the value of the $LCER$ register (Section 3) and it is set in the corresponding entry of the LSIQ (the $CBE$ bit) as soon as that entry is allocated in the decode stage. Then, according to the value of the $CBE$ bit, the load is issued ($CBE$ bit equal to 1) or blocked ($CBE$ bit not set).

Once a low confidence branch prediction is resolved, the blocked loads will be issued to memory if the prediction was correct. Otherwise, if the branch has been mispredicted, the state recovery mechanism already implemented in the processor will squash those loads. Note that for wrong estimates (a branch is actually well predicted), only a small logic that resets the $CBE$ bits must be provided, since the rest of the recovery mechanism remains unmodified.

## 3.3 Control-Independent Loads

As described above, once our mechanism classifies a hard-to-predict branch as low confidence, it blocks all the following loads until the branch is resolved. To further analyze the effects of both control dependent and control independent loads, we present a simple modification of the mechanism that consists in not blocking the control-flow independent loads beyond the point where both paths of the low confidence branch re-converge. The aim of this proposal is to prefetch memory accesses that would be done later during the correct path execution. The possible benefits are twofold. On the one hand, when a low confidence branch is actually mispredicted, the control independent loads may have generated useful prefetches. On the other hand, when a low confidence branch is actually a correctly predicted branch, not all the right path loads have been blocked, reducing the potential performance degradation.

To do this modification, we need to increase the amount of semantic information available for our mechanism, making it possible to detect loads that are control-independent on a low confidence branch. Hence, we include two simple new tasks to our mechanism to detect post-reconvergent independent loads. Firstly, a heuristic to predict the reconvergent point, this is the first instruction present in all possible branch outcomes, associated to a conditional branch. Secondly, a way to track data dependences among the source registers of instructions. For the former, the heuristic is the same used by the Skipper mechanism [3], based on control-flow code patterns generated by compilers for normal conditional structures (*if-then-else* and *loop conditional*). For the latter, to handle instruction dependencies, our mechanism uses a bit-mask vector. Each entry of this vector is a bit associated to each logical register. At the decode stage, once a branch is identified as low confidence ($LCER$=1) and while the reconvergent PC predicted by the heuristic is not found, each instruction sets to 1 the bit associated with its destination register in the mask vector. When the reconvergent instruction is decoded, instructions check the vector mask entries corresponding to its source registers. If none of these entries are set to 1, the instruction is marked as control-flow data independent, in other case, it is control dependent. These control dependent instructions continue updating the corresponding destination vector entry, doing the logic OR with the value of the vector mask entries corresponding to their source registers.

## 4. EXPERIMENTAL FRAMEWORK

The results in this paper have been evaluated using a cycle-accurate execution-driven simulator based on SMT-SIM [20]. This simulator lets us model an aggressive single-thread processor with wrong-path execution and includes an enhanced memory hierarchy implementation, with bus and port contention, bank conflicts, and support for MSHRs management.

Table 2 lists the main configuration parameters of our simulation setup. To model a large instruction window processor, we scale up the main processor structures (ROB, issue queues, and register file) with respect to a normal superscalar processor. Among others, the processor has a 1024-entry reorder buffer, 1024 registers, and 512-entry issue queues. Also, we use an issue width of 4 instructions per cycle, since it is a good trade-off between performance and energy. For generality, this strategy provides a good approximation for large instruction window proposed trends. We leave the particular implementation unspecified, either [6] or [19] implementations can be utilized.

We have choosen the perceptron branch predictor [9] because it is one of the most accurate branch predictors in the literature. We have analyzed different perceptron setups and selected the most efficient for the configurations evaluated (see Table 2). Note that the better the branch predictor is, the less the load blockade mechanism is started, and the less misspeculated memory instructions are executed.

| **Processor core** | |
|---|---|
| Issue width | 4 |
| Reorder buffer size | 1024 |
| INT/FP registers | 1024 / 1024 |
| INT/FP/LS issue queues | 512 / 512 / 512 |
| INT/FP/LdSt units | 4 / 4 / 2 |
| Perceptron Branch predictor | 256 perceptrons<br>4096 x 14 bit local and<br>40 bit global history |
| Branch mispred. lat. | minimum 6 cycles. |
| RAS | 64 |
| **Memory subsystem** | |
| Icache | 64 KB, 4-way, 1 cyc latency |
| Dcache | 64 KB, 4-way, 3 cyc latency |
| L2 Cache | 1 MB, 8-way, 16 cyc latency |
| Caches line size | 64 bytes |
| Main memory latency | 500 cycles |

**Table 2: Large Instruction Window processor configuration**

Our execution-driven simulator emulates Alpha standard binaries. All experiments were performed with the SPEC 2000 Integer benchmark (SPECint) suite [18]. The benchmarks were compiled with the Compaq C V5.8-015 compiler on Compaq UNIX V4.0 with the -O3 optimization level. In order to reduce simulation time, we run 300 millions representative instructions for each benchmark using the reference input set. To identify the most representative simulation segments we have analyzed the distribution of basic blocks as described in [15].

## 5. EVALUATION

In this section, we evaluate our speculative load control technique. The main objective of our analysis is to show the ability of our mechanism to reduce the amount of speculative loads and instructions executed in a large instruction window processor. We also show the performance achieved when this mechanism is applied.

### 5.1 Reduction of Misspeculated Loads

Figure 4 shows the total amount of wrong-path load instructions executed with our 1024-entry instruction window model. The left-most bar shows the total number of misspeculated load instructions executed by each model in normal operation, that is, without applying the speculative control mechanism. The second bar represents the reduction in the number of loads when our Speculative Control Mechanism (SCM) with the Decoupled Counters confidence estimator is used. The third bar shows the SCM with the modification to take into account control-independent loads (SCM_ldCI). Finally, the right-most bar shows the results for the speculative control mechanism when a perfect confidence estimator is applied (SCM_Perfect). This perfect confidence estimator detects branch mispredictions at the fetch stage, always classifying them as low confidence. We add this last bar to show how close our mechanism is respect to the potential upperbound.

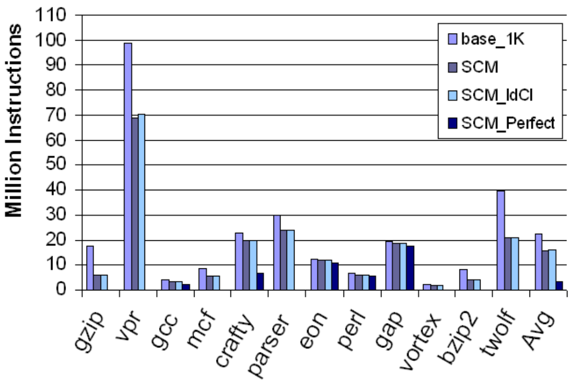

**Figure 4: Speculative Loads Reduction for 1K-entry ROB processor**

This 1024-entry instruction window processor achieves an average reduction of near 7 million loads over a total of 23 millions. The percentage of misspeculated load instructions avoided is between 4% (*eon*) and 67% (*gzip*) and the average is 26%. In particular, the number of reduced wrong-path loads is significant for benchmarks *vpr* and *twolf* in which the number of loads are reduced by 30 and 20 millions respectively. These benchmarks execute an important number of wrong-path loads per mispredicted branch: 42,7 loads for *vpr* and 24,4 for *twolf*.

As expected, our speculative control mechanism performs better on benchmarks with a large number of branch mispredictions, since the opportunities to block loads are higher. For example, the benchmark *vpr* presents 2,3 millions of mispredicted branches (9,1% of misprediction ratio), from which, 886.127 (38%) are detected as low confidence branches, reducing the number of misspeculated loads by 30 millions.

As we have described, the SCM_ldCI version of the speculative control mechanism do not avoid the issue of load instructions identified as control independent under a low confidence branch. With this modification, the speculative control mechanism lets issue around 100,000 loads instruction more on average with respect to the mechanism without this variation. Only *vpr* benchmark has a remarkable number of control independent loads detected and issued (1,4 millions).

Nevertheless, even if the perfect confidence estimator is applied, some benchmarks still execute an important amount of wrong-path loads. This is due to mispredictions caused by indirect jumps whose addresses are wrongly predicted. However, this misses are not avoidable with the confidence estimator because address computation is not a binary condition. For this reason, the benchmarks *eon*, *perl* and *gap* do not show a remarkable wrong-path load reduction with our mechanism, since conditional branches are not a significant problem for them.

## 5.2 Reduction of Misspeculated Instructions

Due to dependences among instructions, loads are not the only kind of instructions blocked by our approach. As the speculative control mechanism blocks the low confidence loads in the LSIQ, the instructions dependent on a blocked load are not issued because some of their source operands are not available. Then, this dependent instruction reduction contributes to a major energy saving, since they will may not be executed.

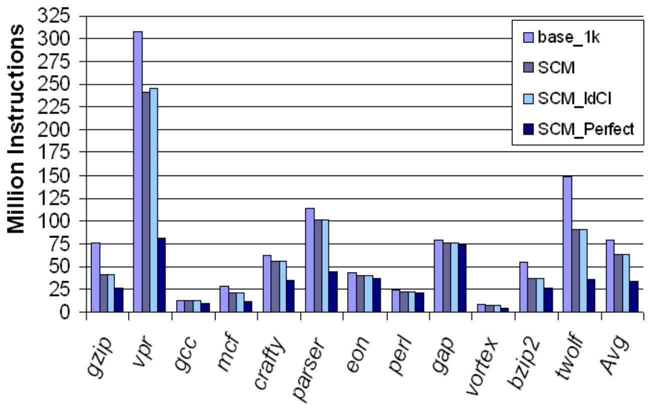

**Figure 5: Wrong-path instructions Reduction for 1K-entry ROB processor**

Figure 5 shows the total number of wrong-path instructions executed for the 1024-entry instruction window processor. The left-most bar represents the total number of wrong-path instructions for normal execution, the second bar (SCM) shows the total number of wrong-path instructions when the SCM is applied, the third bar is the SCM_ldCI modification, and the last bar (SCM_Perfect) depicts the oracle situation when the perfect confidence estimator is applied.

The SCM reduces 18,5 millions of misspeculated instructions for the 1024-entry instruction window processor, which represents an average of 18% reduction of wrong-path instructions. The mechanism with the control independence variation (SCM_ldCI) obtains similar reduction respect to the normal SCM. On average, this second proposal is executing only 300,000 instruction more over approximately 66 millions of wrong-path instructions in total.

## 5.3 Performance

Figure 6 shows the IPC variation normalized to the 1024-entry instruction window processor baseline obtained with the initial speculative control mechanism (SCM) version and the modified SCM_ldCI version. Also, the performance improvement with the perfect confidence estimator is shown (SCM_Perfect). The average IPC (Hmean) is calculated as the harmonic mean of the IPC values for benchmarks.

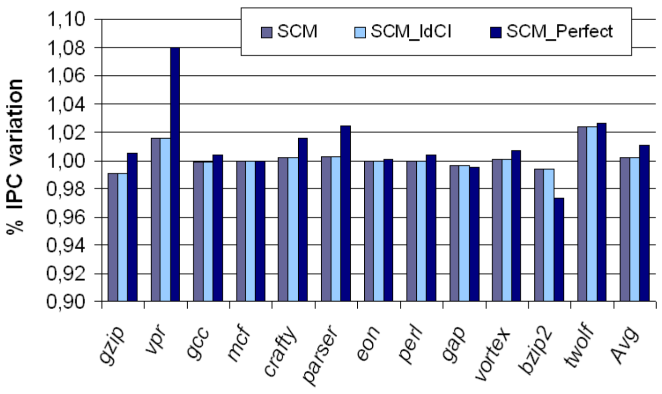

**Figure 6: Performance variation for SPECint for 1K-entry ROB processor model**

Figure 6 shows that, on average, our speculative control mechanism (SCM) with a realistic confidence estimator does not cause any performance loss. On the contrary, there is a moderate 2% performance improvement for two benchmarks (*vpr* and *twolf*). There are only two benchmarks (*bzip2* and *gzip*) that suffer from negligible performance degradation (below 0.1%). Besides, *bzip2* loses performance even with SCM_Perfect, since wrong-path references are beneficial for this benchmark (see Figure 3).

From Figure 6 another lecture can be obtained. The control-independent loads detected down the wrong path do not improve performance. These performance results can be explained by the data show in Table 3. This table shows the distribution of first level data cache and second level cache misses. For all cache misses (Total), the table shows which misses are caused down the wrong-path (WP) and which of them are produced by control-independent accesses (CI). The number of cache misses produced by control-independent instructions is very low for both caches. These data indicate that most of the control-independent accesses are hitting in the caches, and thus there is not any opportunity to improve performance through prefetching. In other scenarios in which the branch misprediction is higher or the L2 cache latency is larger, the opportunities to benefit from control-independent loads could be higher, since more instructions are executed down the wrong path.

It can be concluded from these performance results that our simple control speculative mechanism (SCM) is an effective way of reducing the number of wrong-path loads (26% on average) without any performance penalty. Therefore, since overall energy consumption is dependent on performance and power ($Energy = Power * Performance$), our

| | DCACHE Misses | | | L2CACHE Misses | | |
|---|---|---|---|---|---|---|
| **Spec** | **Total** | **WP** | **CI** | **Total** | **WP** | **CI** |
| gzip | 2.044.396 | 221.702 | 277 | 179.632 | 145.679 | 9 |
| vpr | 6.616.353 | 2.381.658 | 33.228 | 1.298.579 | 559.569 | 5.944 |
| gcc | 9.628.087 | 195.559 | 610 | 158.140 | 46.702 | 66 |
| mcf | 18.499.078 | 1.174.174 | 1.713 | 17.054.411 | 768.106 | 700 |
| crafty | 594.977 | 141.681 | 894 | 30.012 | 8.428 | 72 |
| parser | 2.872.756 | 391.260 | 1.602 | 776.244 | 93.981 | 170 |
| eon | 63.757 | 13.390 | 0 | 9.707 | 8.320 | 0 |
| perl | 168.060 | 41.000 | 69 | 65.956 | 14.047 | 37 |
| gap | 403.345 | 69.380 | 0 | 366.970 | 42.670 | 0 |
| vortex | 1.342.334 | 196.196 | 798 | 316.857 | 30.588 | 599 |
| bzip2 | 483.503 | 86.972 | 16 | 88.742 | 17.042 | 2 |
| twolf | 5.588.594 | 1.070.673 | 629 | 1.060.027 | 180.209 | 365 |
| Avg | 2.709.651 | 437.225 | 3.466 | 395.533 | 104.294 | 660 |

**Table 3: Caches Misses Distribution with Control Independent Accesses**

mechanism effectively involves energy saving, because it decreases dynamic power by reducing the number of wrong-path instructions executed without losing performance. We have evaluated the energy consumption of the caches and the confidence estimator using the CACTI tool [16].We modified CACTI to model a tagless branch predictor and similar structures (like our confidence estimator), and to work with a setup expressed in bits instead of bytes. Assuming a 0.10$\mu$m technology and currently achievable frequencies, we compute the energy consumption of the cache hierarchy and branch predictor for both the baseline and the SCM approach. For the later, the additional consumption due to the confidence estimator is also considered. Only taking into account the power comsuption of those structures, our mechanism achieves an average reduction in energy of 6,5%.

## 6. RELATED WORK

The effects of wrong-path execution on processor performance have been analyzed in several works. Many of these studies find that wrong-path references are beneficial for performance in some programs, but detrimental in some others. Combs *et al.* [4] reported that wrong-path references are slightly beneficial for performance, increasing it by 1% on average. On the contrary, it is shown in [2] that wrong-path memory references are more likely to pollute the caches. We can deduce from these studies that the impact of wrong-path execution is highly dependent on the particular programs evaluated.

According to their results, Bahar *et al.* [2] propose using a separate associative buffer (*confidence buffer*) to avoid the pollution effects of mispredicted references. This mechanism uses a confidence estimator to indicate when the processor is likely to be executing wrong-path instructions. In this case, the value of all wrong-path memory references is stored in the separate buffer. A similar proposal, namely the *Wrong-Path Cache*, is presented by Sendag *et al.* [14]. This cache and the conventional first level cache are always accessed in parallel to look for memory data.

Mutlu *et al.* [12] do a deep analysis of the impact of wrong-path memory references on current out-of-order and RunAhead execution processors. They find that L2 cache pollution is the most significant effect of wrong-path references. Based on this study, they propose in [11] using the first level caches as filters to reduce the second-level cache pollution caused by speculative memory references, including both wrong-path and hardware prefetcher references. This work focus on performance and unlike other works, Mutlu *et al.* do not use a confidence estimator. They establish that mispredicted branches are resolved before 94% of wrong-path L2 misses complete. Therefore, whether or not an L2 cache miss is speculative is usually known before the block is placed into the L2 cache.

However, only few works have focused on the impact of energy consumption of speculated wrong-path instructions. Energy-aware techniques usually sacrify design flexibility or performance to reduce energy. Manne *et al.* [10] observe that execution down the wrong-path results in a significant increase in the number of executed instructions, which they called *extra work*. They propose *pipeline gating* to reduce this extra work and decrease the overall energy consumed by the processor with a slight performance loss.

Regarding confidence estimation, most research is related to branch prediction. Initially, Smith [17] proposes assigning confidence levels to different counter values in predictors based on saturating counters. Jacobsen *et al.* [8] study later an ideal method to profile branch predictors and propose different uses for branch prediction confidence mechanisms. Also, a detailed analysis of methods and accurate metrics to compare confidence estimation mechanisms are presented in [7].

Finally, regarding control independence mechanisms, Rotenberg et al. present a mechanism to exploit control flow independence in superscalar processors [13]. Their approach is based on identifying control independent points dynamically, and a hardware organization of the instruction window that allows the processor to insert the instructions after a branch misprediction between instructions previously dispatched, i.e., after the mispredicted branch and before the control independent point. Cher et al. present Skipper [3], a mechanism to overlap the latency of hard-to-predict branches with the execution of control-flow independent instructions following the reconvergent point of those branches. To achieve this, when a repeatedly mispredicted branch is detected, the fetch is redirected to the re-convergent point, creating a gap in the reorder buffer (large enough to hold the skipped instructions), and the processor proceeds to execute the instructions after the re-convergent point. When the branch is resolved, the skipped instructions are executed. Dependences among skipped instructions and the instructions after the re-convergent point are checked to ensure the correctness of the execution, performing recovery actions when needed.

## 7. CONCLUSIONS

In this paper, we present a simple speculative control mechanism, called SCM, focused on future high-performance processors with large instruction windows. Our design extends the load-store queue functionality to block load instructions that depend on low confidence conditional branches until they are resolved, preventing misspeculated loads from accessing the memory hierarchy, and thus reducing energy consumption. In order to estimate the confidence of branch predictions, our proposal includes a new confidence estimator implementation that decouples the counters, providing estimations for taken and not taken outcomes.

Our results show that, on average, SCM is able to reduce 26% wrong-path load instructions. In addition, the instructions that depend on blocked loads are also blocked, since not all their source values are available. On average, SCM reduces 18% wrong-path instructions. Moreover, this reduction is achieved without any performance degradation.

We also propose a modification to study the performance improvement of letting the processor to issue control-flow independent loads under an estimated low confidence branch. This study focuses on analyzing the performance benefit of wrong path prefetches. Our studies show that control-independent loads are worthless to consider for our mechanism since most of them hit the L1 data cache and, thus, no useful prefetch is performed.

## 8. ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01 and grant AP2003-3682, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center (BSC-CNS).

## 9. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th annual Intl. Symposium on Microarchitecture*, Washington, DC, USA, 2003.

[2] R. Bahar and G. Albera. Performance analysis of wrong-path data cache accesses. In *Workshop on Performance Analysis and its Impact on Design, ISCA'98.*, 1998.

[3] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th annual Intl. symposium on Microarchitecture*, Washington, DC, USA, 2001.

[4] J. Combs, C. B. Combs, and J. P. Shen. Mispredicted path cache effects. In *European Conference on Parallel Processing*, 1999.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 10th Intl. Symposium on High Performance Computer Architecture*, Madrid, Spain, 2004.

[6] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramírez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3), 2005.

[7] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th annual Intl. Symposium on Computer Architecture*, Washington, DC, USA, 1998.

[8] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th annual Intl. symposium on Microarchitecture*, Washington, DC, USA, 1996.

[9] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th Intl. Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2001.

[10] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th annual Intl. Symposium on Computer Architecture*, Washington, DC, USA, 1998.

[11] O. Mutlu, H. Kim, D. Armstrong, and Y. Patt. Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance. In *16th Symposium on Computer Architecture and High Performance Computing*, 2004.

[12] D. N. A. Onur Mutlu, Hyesoon Kim and Y. N. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54(12), 2005.

[13] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. In *Proceedings of the 5th Intl. Symposium on High Performance Computer Architecture*, Washington, DC, USA, 1999.

[14] R. Sendag, D. J. Lilja, and S. R. Kunkel. Exploiting the prefetching effect provided by executing mispredicted load instructions. In *Proceedings of the 8th Intl. European Conference on Parallel Processing*, London, UK, 2002.

[15] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 Intl. Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2001.

[16] P. Shivakumar, and N.P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. *Research Report 2001/2.*, Western Research Laboratory, 2001.

[17] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual Intl. Symposium on Computer Architecture*, Los Alamitos, CA, USA, 1981.

[18] SPEC. Standard performance evaluation corporation (spec) 2000 benchmark suite. http://www.spec.org/cpu2000/.

[19] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the ASPLOS-XI*, NY, USA, 2004. ACM Press.

[20] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Int. Annual Computer Measurement Group Conference*, 1996.