Speculating to Reduce Unnecessary Power Consumption

ENRIC MUSOLL
Tidal Networks, Inc.

The power consumption of current processors keeps increasing in spite of aggressive circuit design techniques and process shrinks. One of the reasons for this increase is the complexity of the microarchitecture required to achieve the performance that each processor generation demands. These techniques, such as branch prediction and on-chip level two caches, increase not only the power consumption of the committed instructions, but also the useless power associated with those block accesses that generate results that are not needed for the correct execution and commit of the instructions.

In this work, the different accesses that a particular block receives are classified into four different components, based on whether the accesses are performed by instructions of the correct path or the wrong (mispredicted) path, and also based on whether the results of the accesses are needed or not for the correct execution of the instructions. Out of the four components, only one accounts for the useful accesses to the block, that is, accesses performed to correctly execute instructions that will be committed. The other three components account for the useless activity on the block. The simulations performed indicate that, if the useless power dissipation of a high-performance processor could be totally removed with no performance degradation, the overall processor power consumption would be reduced by as much as 65% compared to the same processor in which all the blocks are accessed every cycle.

This work then proposes a microarchitectural technique that targets the reduction of the useless power dissipation. The technique consists of predicting whether the result of a particular block of logic will be useful in order to execute the instructions (no matter whether the instructions will be eventually committed or not). If it is predicted useless, then the block is disabled.

A case example is presented where two blocks are predicted for low power: the on-chip L2 cache for instruction fetches and the branch target buffer (BTB). The IPC versus power-consumption design space is explored for a particular microprocessor architecture. Both the average and the peak power consumption are targeted. High-level estimations are done to show that it is plausible that the ideas described might produce a significant reduction in useless block accesses. As an example, 65% accesses to the L2 cache can be eliminated at a 0.2% IPC degradation, and about 5% accesses to the BTB can be saved at the penalty of 0.7% IPC reduction.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General

General Terms: Design

Additional Key Words and Phrases: Low-power design, low-power microarchitectures

Author's address: Tidal Networks, Inc., 697 River Oaks Parkway, San Jose, CA 95134; email: enric@tidalnetworks.com.

1. INTRODUCTION

Until recently, design techniques for reducing the average and peak power consumption have been applied mainly at the circuit and process levels [Chandrakasan et al. 1992]. However, the power consumption of current and future high-performance processors keeps increasing even with aggressive circuit design techniques and process shrinks [Gwennap 1998a]. The Alpha 21364 processor, for example, is estimated to consume around 100 W [Bannon 1998]. One of the reasons for this continuous power-consumption increase is the complexity of the microarchitecture needed to achieve the performance that each high-performance processor generation requires.

For high-performance processors, only those low-power techniques that do not significantly hinder the performance have been regularly applied. At the architecture level, for example, decreasing the operating frequency of the processor is a common technique when no useful work is done.

When the processor is doing useful work, disabling specific units or blocks of logic (those that will not produce any useful result for the execution) is a more fine-grained architectural technique for low power. However, if other than these blocks are disabled, the IPC may potentially be degraded. For some blocks, perfect knowledge is available to determine when the block must be enabled or disabled. One example is the functional units. If no operations are present on the reservation station associated with a functional unit, then the unit can be disabled until an operation is ready. In this case, the unit block is *not accessed* (and thus its "result" is *useless*) to execute the current instruction.

For some other blocks it is more difficult to know whether they need to be accessed or not. For example, the on-chip level-two (L2) cache. In some high-performance processors, the on-chip L2 cache may be accessed concurrently to the level-one (L1) cache to minimize the L2 cache latency in case there is a miss in the L1 cache.[1] There is no way to know whether the L2 cache result will be useful unless the L1 cache hit/miss outcome is known beforehand. The L2 cache block is, then, *accessed* (because the L1 cache may miss) but usually its result is *useless*. If a block could be accessed only when its result is useful, then no useless power would be consumed.

This work is based on the concept that (a) a block is accessed for the execution of an instruction but its result is sometimes useless and (b) the knowledge of whether its result is useless or not is not known for sure when the block starts to be accessed. A high-performance processor would assume that the result of the block is useful and it would always enable the block; if the result happens to be useless, the result is simply not used (but at the cost of some useless power consumption due to the unnecessary access to the block). On the other hand, a power-consumption conscious processor design would disable the block until it is known whether the result of the block is useful. Only at this point the block would be enabled if the result is known to be useful (but at the cost of paying

[1]This way of accessing the two-level cache system may be quite expensive in area since the L2 cache needs to have at least the same number of ports (bandwidth) as the L1. An example of a current processor with simultaneously on-chip L1 and L2 accesses is the AMD K6-III.

some extra cycles of latency, i.e., those used to figure out whether the result is useful or not).

In this work we present a technique that is based on the concept of *prediction for low power*. A hardware predictor is used to predict whether the result for the next access to a block will be useless or not. If the predictor is that the result will be useless, the block is then disabled (through clock gating methods, for example), thus not consuming unnecessary power in the next access. Such predictor might not be fully accurate: sometimes the result will be predicted useless when it is really not (henceforth named *misprediction when useful*); in this case, the predictor should enable the block and re-initiate the access, increasing then the latency of the result of the block, which potentially reduces the IPC; a decrease in IPC may offset the energy savings obtained with the predictor since more cycles are needed to execute the application. A function will be presented that can be used to evaluate how good a predictor for low power is. This function depends on the accuracy of the predictor in predicting that a result will be useful, and also in predicting that a result will be useless.

The different accesses that a particular block receives are classified into four different components, based on whether the accesses are performed by instructions of the correct path or the wrong (mispredicted) path, and also based on whether the results of the accesses are needed or not for the correct execution of the instructions. Out of the four components, only one accounts for the useful accesses to the block, that is, accesses performed to correctly execute instructions that will be committed. The other three components account for the useless activity on the blocks. The simulations performed indicate that, if the useless power dissipation of a high-performance processor could be totally removed with no performance degradation, the overall processor power consumption would be reduced by as much as 65% compared to the same processor in which all the blocks are accessed every cycle.

A case example is presented where two blocks are predicted for low power: the on-chip L2 cache for instruction fetches and the branch target buffer. The IPC versus power-consumption design space is explored for a particular microprocessor architecture. Both the average and the peak power consumption are targeted. Although the power analysis is beyond the scope of this work, high-level estimations are done to show that it is plausible that the ideas described might produce a significant reduction in useless block accesses.

2. USEFUL AND USELESS ACCESSES

The total amount of accesses (A) to the blocks of a processor may be classified as (a) whether the accesses are performed by instructions of the correct path (A_{cp}) or the wrong path (A_{wp}); (b) whether the results of the accesses are needed (A^n) to execute the instructions (regardless of whether the instructions are finally committed of not) or not needed (A^{nn}). Thus, $A = A_{cp} + A_{wp} = A^n + A^{nn}$.

Moreover, there are block accesses performed by instructions within the correct path that generate not-needed results and, analogously, block accesses performed by instructions within the wrong path that generate needed results for the execution of the instructions (although these instructions will never

commit). Thus, the total amount of block accesses may be broken down as follows:

$$A = \underbrace{A^{n}_{cp}}_{\text{Useful accesses}} + \underbrace{A^{nn}_{cp} + A^{n}_{wp} + A^{nn}_{wp}}_{\text{Useless accesses}}.$$

Out of the four terms of the expression, only the first one represents the total amount of useful accesses since they are needed to correctly execute instructions that will be committed. All the accesses within the wrong path only contribute with useless power dissipation since no instructions will be committed. Note that the A^{nn}_{cp} term also contributes with useless power since the results from the blocks are not used.

2.1 Reducing the Amount of Useless Accesses

The amount of useless accesses can be reduced following two strategies: (a) by reducing the overall number of instructions executed within a wrong path. This will reduce the A^{n}_{wp} and A^{nn}_{wp} terms; and (b) by reducing the amount of useless accesses generated by any instruction fetched into the processor. This will reduce the A^{nn}_{cp} and A^{nn}_{wp} terms.

The first strategy is well known in the microarchitecture community since the objective is to improve the branch predictor. The more accurate this predictor is, the smaller the number of instructions will be executed within a wrong path and, therefore, the lower the A^{n}_{wp} and A^{nn}_{wp} terms will be.

The second strategy is the one tackled in this work. In addition, two patents [Kennedy and Croxton 1998; Jaggar 1996] and the work in Manne et al. [1998] have focused on reducing the useless power due to speculative instructions that are fetched by the processor but they are never committed. These instructions access different blocks of the processor until they are flushed out of the pipelines. All the three works are based on a confidence estimation of the prediction performed by the branch prediction mechanism.

The technique patented in Jaggar [1996] disables the branch predictor based on some information provided by the branch predictor itself in the previous access. This information is a prediction of when the next branch instruction will happen; therefore, between the previous access to the branch predictor and the time that the next branch instruction is fetched, no accesses to the branch predictor are needed (provided that the prediction for low power was correct), thus the branch predictor block may be disabled.

The mechanism patented in Kennedy and Croxton [1998] also disables the branch predictor based on a prediction. In this case, however, this prediction (which is made based on a limited history of taken/not-taken outcomes of the branches) is performed by a different block than the branch predictor itself.

The work in Manne et al. [1998] is a more general technique in the sense that blocks other than the branch predictor are disabled based on a confidence estimation [Grunwald et al. 1998; Jacobsen et al. 1996] of the prediction performed by the branch predictor. Thus, when the branch prediction is detected to have a high probability of mispredicting a branch, potential wrong-path instructions are prevented from entering the pipeline. The results show that up

to a 38% reduction in wrong-path instructions may be achieved with a small degradation of the processor performance.

The fundamental difference between Manne et al. [1998] and the methodology presented in this work is the type of unnecessary block accesses that are eliminated: in Manne et al. [1998], only those useless accesses performed by the wrong-path instructions are targeted (i.e., components A^{n}_{wp}, A^{nn}_{wp}) whereas in this work the goal is to also eliminate these useless accesses performed by instructions of both the correct and wrong paths (i.e., components A^{nn}_{wp}, A^{nn}_{cp}). Thus, the methodology explained here may be applied along with the technique in Manne et al. [1998] for even greater power savings.

2.2 Upper-Bound Useless Energy Dissipation

In this section, a case example is presented to provide a first-order estimation of the overall useless energy used by a high-performance microprocessor. Only those power-hungry blocks common to current high-performance microprocessors will be taken into account. To simplify the analysis, the following assumptions are made (which, for the high-level power estimation targeted in this work, we believe that are reasonable to take):

—*An access to a block takes one processor cycle*. This assumption is valid for the purpose of this work even if the block is fully pipelined (in this case, however, the latches between stages would be assumed to consume no power). To account for a nonfully-pipelined block, the throughput of that block should be considered in the analysis.

—*The dynamic dissipation component accounts for the total power consumption of the block*.

—*The power consumption of a block is proportional to the number of accesses it receives*. This assumption implies that (a) clock gating or other mechanism is implemented to shut down the block when it is not accessed; (b) if a block is accessed twice in a given cycle (because the block has two or more ports), its power consumption will be twice as much as the case in which only one port is used; and (c) the power consumption of a block can be coarsely estimated without taking into account the particular value at the inputs of the block.

With these assumptions, the total energy (E_T) of the microprocessor is modeled as

$$E_T = \sum_{i=1}^{B} (W|_{b_i} \times A|_{b_i}) \tag{1}$$

where B is the number of blocks, $W|_{b_j}$ is the power consumption of block j, and $A|_{b_j}$ is the total number of accesses to block j. The expression (1) can be further broken down as

$$E_T = \sum_{i=1}^{B} (W|_{b_i} \times (A^{n}_{cp}|_{b_i} + A^{nn}_{cp}|_{b_i} + A^{n}_{wp}|_{b_i} + A^{nn}_{wp}|_{b_i})). \tag{2}$$

Table I. Access Type: Needed Versus Not-Needed

Block	A^n	A^{nn}
L1 instruction cache	#code look-ups AND hit, plus line replacements and invalidations	#code look-ups AND miss
L1 data Cache	#load and store accesses AND hit, plus line replacements, write-backs and invalidations	#load and store accesses AND miss
L2 unified cache	#code look-ups and #load and store accesses AND (L1 miss AND L2 hit), plus line replacements, write-backs and invalidations	#code look-ups and #load and store accesses AND (L1 hit OR L2 miss)
Instruction TLB	#code look-ups AND hit, plus replacements and invalidations	#code look-ups AND miss
Data TLB	#load and store look-ups AND hit, plus replacements and invalidations	#load and store accesses AND miss
BTB	#look-ups AND (hit AND taken), plus updates	#look-ups AND (miss OR not taken)
Register File	#register reads and writes except register bypasses	#register bypasses
Simple Int. FU	#integer instr. except NOPs, Multiplications and Divisions	#NOPs
Simple FP FU	#FP instr. except Multiplications and Divisions	NA
Complex Int. FU	#integer Multiplications and Divisions	NA
Complex FP FU	#FP Multiplications, Divisions and SQRTs	NA

If all the blocks are accessed every cycle, the total energy (which will be the theoretical maximum energy dissipated) is

$$E_T|_{\mathrm{MAX}} = \#\mathrm{Cycles} \times \sum_{i=1}^{B} (W_T \times w_{b_i}) = \#\mathrm{Cycles} \times W_T$$

where w_{b_j} is the contribution of block j to the overall power consumption (W_T). Thus, E_T in Equation (2) may be re-written as

$$W_T \times \sum_{i=1}^{B} (w_{b_i} \times (A^n_{cp}|_{b_i} + A^{nn}_{cp}|_{b_i} + A^n_{wp}|_{b_i} + A^{nn}_{wp}|_{b_i})). \quad (3)$$

Table I lists the microarchitecture blocks considered in this case example and, for each of the blocks, it explains when a needed and not-needed access is performed (regardless of whether the instruction that generates the access belongs to within a correct path or not).

2.3 Results

The amount of total accesses to each of the blocks have been obtained by executing several applications of the SPEC95 benchmark on the SimpleScalar tool set [Burger and Austin 1997]. The default microarchitectural parameters of the SimpleScalar out-of-order simulator have been used in this work. Some of these very briefly are, 8 KB DM 1-cycle Inst. L1, 8 KB DM 1-cycle data L1, 256 KB 4-way 6-cycle Unif. L2, 64-entry 4-way Inst. TLB, 128-entry 4-way data TLB, 2K-entry 4-way BTB, bimodal branch predictor, 4 simple Int. FUs, 1 complex Int. FU, 4 simple FP FUs, and 1 complex FP FU.

Table II. Number of Accesses/Number of Cycles Ratios

Block	SPEC95			
	A^n_{cp}	A^{nn}_{cp}	A^n_{wp}	A^{nn}_{wp}
IC	0.47	0.04	0.06	0.00
DC	0.37	0.01	0.03	0.00
UC	0.07	0.77	0.00	0.08
IT	0.46	0.00	0.05	0.00
DT	0.35	0.01	0.03	0.00
BT	0.29	0.03	0.05	0.00
RF	2.40	0.75	0.22	0.11
SI	0.96	0.04	0.10	0.00
CI	0.01	NA	0.00	NA
SF	0.08	NA	0.01	NA
CF	0.05	NA	0.00	NA
Total	5.51	1.65	0.55	0.19

IC—2-p L1 ICache
DC—2-p L1 DCache
IT—2-p ITLB
DT—2-p DTLB
RF—6-p Reg. File
SI—4 SInt FUs
SF—4 SFP FUs
CF—1 CFP FU
UC—2-p L2 UCache
BT—2-p BTB
CI—1 CInt FU

Table III. Different Sets of Power Ratios

Block Group	Block Port/Instance	Power Ratios			
		A	B	C	D
Clock System		0.20	**0.50**	0.12	0.12
Caches	IC	0.05	0.03	**0.08**	0.03
	DC	0.05	0.03	**0.08**	0.03
	UC	0.08	0.05	**0.10**	0.04
	IT	0.02	0.01	**0.02**	0.01
	DT	0.02	0.01	**0.02**	0.01
	BT	0.03	0.02	**0.04**	0.01
FUs	RF	0.005	0.005	0.005	**0.025**
	SI	0.005	0.005	0.005	**0.015**
	CI	0.035	0.015	0.015	**0.07**
	SF	0.035	0.025	0.025	**0.065**
	CF	0.075	0.035	0.035	**0.10**

Table II shows the values obtained for different terms described in Section 2 (A^{nn}_{cp}, A^n_{cp}, A^{nn}_{wp}, A^n_{wp}), for both benchmarks, for each of the blocks, and for the four types of accesses. The values are relative to the total number of cycles needed to execute the applications. These ratios may be larger than 1 (e.g., in the *register file*) in case the block is frequently accessed and it has several ports. The *L2 unified cache* and the *register file* are the blocks that present a larger number of useless accesses for both benchmarks.

Table III shows four different sets of the w_{b_j} power ratio values. These values are shown per basic instance of a block; thus, if block b_j is n-ported or there are n instances of it, the ratio value associated with this block is $n \times w_{b_j}$. Each set stresses different parts of the processor: set B has the highest power consuming clock system, set C features power hungry caches, set D stresses the functional units, and set A is an average of the other three sets. The values in the table

have been partially obtained from some published data [Burd and Peters 1994; Dobberpuhl et al. 1992; Gowan et al. 1998; Meng et al. 1995; Santhanam 1996; Scott et al. 1998; Tiwari et al. 1998] and from extrapolations when no information is available.

In this study, the local clocks of the different blocks are assumed to have the capability of being disabled when useless accesses are performed to the blocks. Moreover, the power consumption of the local clock of a block is considered to be proportional to the power of the block, and the power of the local clocks of the different blocks adds up to the total power of the clock system.

The conclusion of this study is that, if the useless energy dissipation could be totally removed at the cost of no IPC degradation, the overall energy consumption would be reduced to about 65%. Note that the elimination of the not-needed accesses to the blocks performed by instructions within the correct path usually comes at the cost of increasing the total amount of cycles needed to run the application. These extra cycles will increase the amount of useless energy dissipated and, therefore, they may offset the energy savings previously obtained.

Other cases with different amount of energy dissipation lie in between the maximum and minimum energy dissipation scenarios. For example, in the case where blocks are accessed only when the instruction requires them, by totally eliminating the useless energy, the overall power consumption of the processor (again, assuming no IPC degradation) may be reduced to about 30%. Note that the 30% is obtained when the useless accesses (energy) are eliminated but in the scenario in which the block is only accessed when the instructions being executed require the accesses (these instructions may belong to either the correct or wrong path). 65% is obtained when the useless accesses are eliminated in the scenario in which the block is accessed in all the cycles, no matter what instruction, if any, is executed. For more details about this results, please refer to Musoll [2000].

3. PREDICTING THE USEFULNESS OF A RESULT

The purpose of the technique presented here is to work around the fact that sometimes the result of the block is not known to be useful before the block is accessed, and this is done through a hardware predictor. If the predictor predicts that the result for the next access to the block is useless, the block will be disabled thus not consuming unnecessary power in the next access. Since the predictor will not be totally accurate, sometimes the result will be predicted useless when it is really not (henceforth named *misprediction when useful*); in this case, the predictor should enable the block and re-initiate the access, increasing then the latency of the result of the block, which potentially reduces the IPC; a decrease in IPC may offset the energy savings obtained with the predictor since more cycles are needed to execute the application. Also, depending on the complexity of the predictor itself, the power consumption of the predictor might offset some of the reductions obtained. Throughout this work, there is the implicit assumption that the predictor consumes an insignificant percentage of the power saved, and therefore the power consumption of the predictor is not taken into account in the simulations. Other nondesired consequences of using

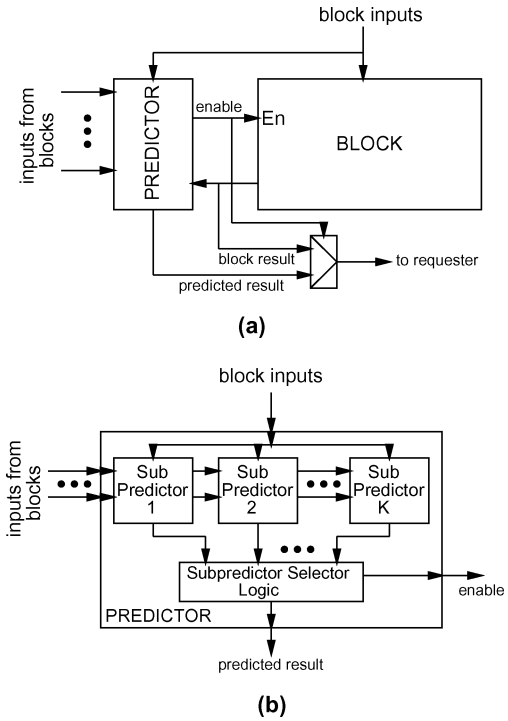


Fig. 1. (a) A generic predictor for low power and its associated block; (b) subpredictors within a predictor.

prediction for low power is the increase in area, validation time, and circuit design issues such as propagating the enable/disable signal to the predicted block. These issues, although important, are out of the scope of this work.

The predictor may also have the functionality of providing a prediction of the result itself (apart from its usefulness) of the block. If this is the case then two situations may happen: (a) the predictor predicts that the result of its block is useless: the block will be disabled, and (b) the prediction is that the result is useful: the predicted result is used and the block is also disabled.

A predictor for low power may be designed in such a way that if a predicted result is given, it is the actual result that the block would provide. An example of a predictor of this kind is a tiny, level-zero (L0) instruction cache. An L0 instruction cache is another level in the memory hierarchy with the purpose of freeing the unified L1 cache from instruction accesses from the processor so that the L1 cache ports are available for data load and store accesses. The L0 cache would predict always that the result of the L1 is useless, thus effectively disabling the L1 cache for instruction accesses unless there is a miss in the L0 cache. Thus, power consumption is saved since the L0 is a less power-hungry block than the larger L1 cache. A study of the power savings using an L0 cache was done in Kin et al. [1997].

A schematic of how a generic predictor is connected to its associated block is shown in Figure 1(a). Potentially, the predictor may have the same inputs

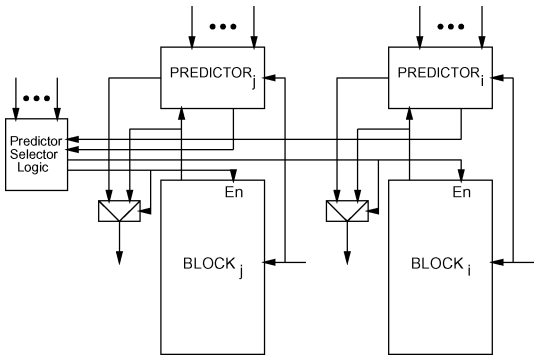


Fig. 2. Selection of the block(s) to be disabled.

as its associated block (plus some other inputs coming from different blocks of the processor) that will be used to make the prediction. Since the predictor may provide the (predicted or actual) result of the block to the requester, a multiplexer is needed as shown in the figure.

3.1 Subpredictors and Multipredictors

A predictor may be composed of several subpredictors as shown in Figure 1(b), each making its prediction based on different information. In this case, a selection of which of the subpredictors will provide the prediction for the block is needed if more than one of the subpredictors predicts that the block result will be useful and at least one of these provides a predicted result of the block. This selection may be static (there is a fixed priority among the subpredictors) or dynamic (where the recent past history of decisions for each subpredictor is kept and used to select the best subpredictor at any time, as done in a combined branch predictor [McFarling 1993]).

Several different blocks of the processor may have predictors associated with them (see Figure 2). Each predictor will make the prediction in a different way since each block functions differently. Some logic (named *predictor selection logic* in the figure) may determine what block(s) to be disabled based on the outcome of the different predictors and possibly on some other inputs from other blocks of the processor. The purpose of this logic is to select a few blocks to be disabled: whenever several of the predictors predict the results to be useless, this logic may disable only a few of the blocks (again those least sensitive to the IPC or/and most power hungry). This helps not degrading too much the IPC by preventing too many blocks from being disabled.

On the other hand, the logic may help in reducing the peak power consumption: whenever none of the predictors predicts that the result of its associated block is useless, this logic may force at least one of the blocks to be disabled (the least sensitive to the IPC or/and the most power-hungry block). This helps reducing the peak power consumption but adversely affects the IPC in the case where all the predictors predict the result to be useful and the predictions where correct.

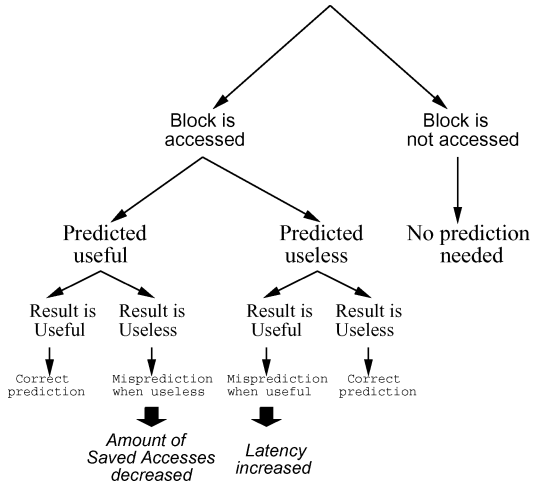


Fig. 3. Consequences of mispredictions.

Care has to be taken in not increasing the critical path of the processor. As mentioned earlier, the predictor is not perfect and therefore it will have an impact on the performance of the processor. This performance degradation would be even greater if the critical path is increased. The predictor logic does not belong to the critical path of the processor unless the delay of the slowest input that the predictor uses to make the prediction plus the delay of the predictor itself is larger than the processor cycle time.

4. PREDICTOR EFFECTIVENESS

The effectiveness of a predictor for low power depends on the accuracy of the predictor in predicting that a result will be useful, and also in the accuracy in predicting that a result will be useless. When a predictor is not ideal in any of these two accuracies, some degradation occurs: as shown in Figure 3, a misprediction when useless decreases the number of accesses to the block that could have been saved; moreover, a misprediction when useful increases the latency of the result of the blocks and this may potentially decrease the IPC of the processor.

The following function is proposed to have a single measure of the performance degradation of a predictor for low power, with respect to the case of perfect prediction:

$$D = \frac{(UsefulAccesses \times MispredUseful) + \alpha(UselessAccesses \times MispredUseless)}{Total\ Accesses}$$
$$= (1 - M) \times MispredUseful + \alpha \times M \times MispredUseless$$

where $MispredUseful$ ($MispredUseless$) is the fraction of useful (useless) accesses that are mispredicted, M is the fraction of useless accesses, and α is a factor that relates the penalty of both types of mispredictions.

Thus, M and α are parameters that depend on the block targeted by the predictor and the application being run on the processor.

4.1 Example

To illustrate how the previous function can be used, several different predictors targeting the same block will be compared for several values of α and M. The block in this example is the L1 data cache.

An L1 data cache hit/miss predictor can save power in the on-chip L2 data cache. Integrating the L2 cache on-chip has been one of the latest techniques to increase the performance of the microprocessor [Gwennap 1998b]. With the L2 cache on chip, two alternatives exist for the time when the L2 cache is accessed:

—access the L2 cache simultaneously with the L1 cache, irrespectively of whether there will be a miss in L1. This reduces the effective access time of L2 when there is a miss in the L1 cache. However, this approach has the following disadvantages:
  - It is quite expensive in area since the L2 cache needs to have at least the same number of ports (bandwidth) as the L1 cache. Part of this bandwidth is wasted when one or more of the accesses is a hit in L1.
  - It results in a high, useless power consumption in the L2 cache when the result is provided by the L1 cache since an access to the L2 is already in progress when the hit/miss outcome from the L1 cache is determined. At this point, the useless access in progress in the L2 could be squashed (some logic needs to be implemented to perform this task) or just allowed to finish and the L2 result is then discarded (this is the worst scenario in terms of power consumption). In any case, the L2 consumes power for at least as many cycles as the L1 takes to compute the hit/miss outcome.

—access the L2 cache only when there is a miss in the L1 cache (thus disabling the L2 cache while determining the L1 hit/miss outcome). This results in a larger effective access time of the L2 cache, but reduces the L2 bandwidth to only the actual number of L1 misses and has no useless power consumption.

An L1 hit/miss predictor provides a trade-off between the two alternatives by accessing the L2 cache only if a miss is predicted in the L1. Mispredictions have different consequences. If the L1 access was predicted to be a

—miss, but it turns out to be a hit, an unnecessary access is performed to the L2, thus consuming both power and an L2 port.

—hit, but it really is a miss, the effective latency of the data (assuming an L2 cache hit) is increased by the L1 latency of the hit/miss determination.

Correct predictions provide the smallest L2 cache latency when the L2 is really needed to be accessed, the lowest L2 energy consumption by preventing an L2 access when its result is useless, and the most efficient use of the L2 bandwidth.

Note that for the L1 data cache, M (i.e., the cache miss rate) is small; therefore, the misprediction when useful (i.e., the misprediction of the access to the cache resulting in a hit) has a larger effect on the degradation, unless this is counterbalanced by a large value of α. Consequently, for this particular block, it is more important to have a very good prediction of hits rather than of misses. On the other hand, because of the larger number of hits, it is simpler to predict

them, and as it will be seen, the predictors vary mostly on the quality of prediction of misses.

A total of six families of L1 predictors are compared. The first three families are either trivial predictors or have already been proposed:

—`Trivial`: either predict always hit or always miss.

—`Saturated Counter` [Kessler et al. 1998]: this predictor consists of a saturated 4-bit counter that tracks the hit/miss behavior of recent data reads. The counter decrements by two on cycles where there is a read miss, and increments by one when there is a hit. The mostsignificant bit of the counter is used to do the prediction.

1,2 and 4-bit counter widths will be evaluated.

—`Per-address Two-level` [Yoaz et al. 1998]: this predictor is based on the concept widely applied in branch predictor theory, in which the address of the load instruction is used to perform the prediction.

The predictors evaluated here store, in an N-entry tagless table, the past H hit/miss outcomes per instruction address. The least-significant bits of the instruction address are used to index this table (first level) to obtain the history vector, which is used (second level) to access a global table of 2^H 2-bit saturated counters that perform the prediction.

The rest of the predictor families evaluated (see Musoll and Lang [2002] for the details) are based on the number of consecutive cache accesses that resulted in a hit since the last miss, and this number is called *distance*:[2]

—`Static Window-based`: this predictor does not keep specific per-distance information and predicts a miss if the distance is less or equal to a threshold. That is, a window is defined and the threshold corresponds to the window size.

Different window sizes will be evaluated.

—`Per-distance Two-level`: it combines the per-distance approach with the two-level mechanism of storing hit/miss outcomes and performing the prediction based on saturated counters. In other words, this family of predictors is the per-distance version of the `Per-address Two-level` family previously described.

—`Per-distance Relaxed Stride`: a dynamic local stride (s) is kept per distance. The stride at distance d is changed whenever a miss happens at that distance. A miss is predicted for an access at distance d if the number of accesses at that distance from the last miss (i.e., the current stride at distance d) is between $s - O$ and $s + O$, where O is a global (i.e., the same for every distance within the table) fixed offset.

The results are shown in Table IV. For each family of predictors, and for each value of α, the predictor with the lowest degradation is shown in italics. The

[2]This concept of distance has been used in a different context in Grunwald et al. [1998] where it is shown that the branch mispredictions occur in clusters, and it is proposed to use the distance from the last mispredict branch as a confidence estimation mechanism [Jacobsen et al. 1996].

Table IV. Degradation Values for Several Instances of α

Predictor Family	Predictor		Percentage of Misprediction H	Percentage of Misprediction M	α 0.05	0.1	0.5	1	2	10	20
Trivial	Always Hit		0	100	**1.0**	**1.0**	*1.6*	*1.8*	*1.9*	*2.1*	*2.9*
	Always Miss		100	0	909	455	142	81	43	9.7	6.7
Saturated Counter	1 bit		2.1	95	20	11	4.4	3.4	2.7	2.2	2.9
	2 bits		2.3	89	22	11	4.6	3.4	2.7	*2.1*	*2.8*
	4 bits		0.06	99	*1.5*	*1.3*	*1.6*	*1.8*	*1.9*	*2.1*	2.9
Per-address Two-level	$H=6$	$N=4$	0.25	70	3.0	1.9	1.5	1.5	1.4	1.5	2.1
		$N=32$	0.22	56	*2.5*	*1.5*	1.2	1.2	1.2	1.2	1.7
		$N=256$	0.26	51	2.9	1.7	1.2	1.1	1.1	1.1	1.5
	$H=8$	$N=4$	0.26	68	3.1	1.9	1.4	1.4	1.4	1.5	2.0
		$N=32$	0.23	54	2.6	1.6	1.2	1.1	1.1	1.2	1.6
		$N=256$	0.26	48	2.8	1.6	*1.1*	1.1	**1.0**	1.1	*1.4*
	$H=10$	$N=4$	0.27	67	3.2	1.9	1.4	1.4	1.4	1.5	2.0
		$N=32$	0.21	53	*2.5*	*1.5*	*1.1*	1.1	1.1	1.2	1.6
		$N=256$	0.24	47	2.7	1.6	*1.1*	**1.0**	**1.0**	**1.0**	*1.4*
Simple Static Window-based	Window of 1		2.1	95	*20*	*11*	*4.4*	*3.4*	*2.7*	2.2	2.9
	Window of 4		6.8	53	63	32	10	6.4	4.0	*1.8*	*2.0*
	Window of 32		24	17	220	110	35	20	11	2.7	2.1
	Window of 256		55	3.0	502	251	78	45	24	5.4	3.8
Per-distance Two-level	$H=6$	$N=4$	0.11	66	*1.7*	*1.2*	1.2	1.3	1.3	1.4	2.0
		$N=32$	0.19	51	2.2	1.4	1.1	1.1	1.1	1.1	1.5
		$N=256$	0.35	49	3.6	2.1	1.2	1.1	1.1	1.1	1.5
	$H=8$	$N=4$	0.11	66	*1.7*	*1.2*	1.2	1.3	1.3	1.4	1.9
		$N=32$	0.17	49	2.0	1.3	**1.0**	**1.0**	**1.0**	1.1	1.5
		$N=256$	0.32	46	3.4	1.9	1.2	1.1	**1.0**	**1.0**	*1.4*
	$H=10$	$N=4$	0.12	66	1.7	*1.2*	1.2	1.3	1.3	1.4	1.9
		$N=32$	0.17	49	2.1	1.3	**1.0**	**1.0**	**1.0**	1.1	*1.4*
		$N=256$	0.32	45	3.4	1.9	1.2	1.1	**1.0**	**1.0**	*1.4*
Per-distance Relaxed Strides	$N=32$	$O=0$	0.43	56	*4.5*	*2.5*	*1.5*	*1.3*	1.3	1.2	1.7
		$O=4$	3.0	35	27	14	4.7	3.0	1.9	**1.0**	1.2
		$O=64$	10	19	93	47	15	8.6	4.8	1.4	1.3
	$N=256$	$O=0$	0.6	53	5.5	3.0	1.6	1.4	*1.2*	1.2	1.6
		$O=4$	4.0	29	36	18	6.1	3.7	2.3	**1.0**	1.1
		$O=64$	12	7.4	106	53	16	9.5	5.2	1.3	**1.0**

lowest degradation across all the families is shown in bold. The misprediction rate (for both hits and misses) is given for each of the predictors. From these results we conclude that the best scheme depends on the value of α. For very low α, the best scheme is `Always Hit`, since the correct prediction of hits is most important. For medium values of α, the best are the `Per-distance Two-level` and `Per-address Two-level` families. Finally, for high α values the best is the `Per-distance Relaxed Stride` family, because of its high correct prediction of misses without large misprediction of hits.

5. OVERALL PROCESSOR POWER REDUCTION

As mentioned earlier, the overall power consumption of a processor with predictors for low power is a function of three factors:

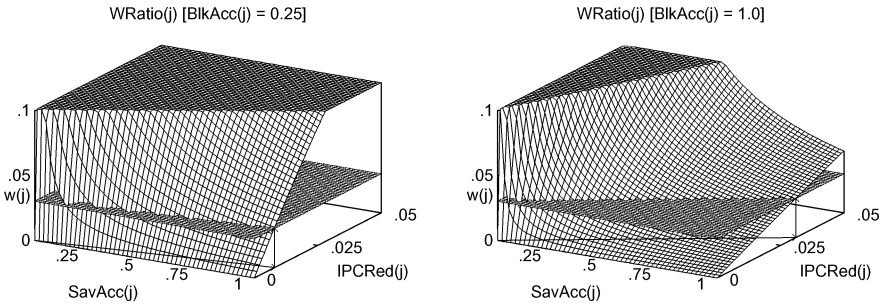


Fig. 4. Design space ($\texttt{WRatio}^j|_{\text{no red}}$) when using a predictor for low power for block j.

—*How accurate each of the predictors is.* The accuracy of a predictor is better analyzed when divided into accuracy when predicting useless (i.e., the predictor predicts useful but the result is actually useless) and when predicting useful (i.e., the predictor predicts useless but the result is actually useful) since the predictor performance in each of these two cases has different consequences. Let $\texttt{SavAcc}^j$ be the percentage of accesses to block j saved by its predictor[3]) and $\texttt{IPCRed}^j$ the percentage of IPC reduction due to results of block_j that were mispredicted when useful).

—*The percentage of the total processor power consumption that corresponds to the blocks being predicted.* Let $\texttt{WRatio}^j$ be this percentage for block j.

—*How sensitive to the IPC the different blocks are.* This sensitiveness is implicit in $\texttt{IPCRed}^j$, and it is also a function of the percentage of times that block_j is accessed (henceforth named $\texttt{BlkAcc}^j$) during the execution of the application.

The relationship among the parameters described above is shown in Figure 4 for a particular block j. To derive the curves plotted in the figure, the following terms are defined:

—E_O: the energy required to execute a given application without any prediction for low power,

—E_P^j: energy required when applying the prediction for block j,

—W^j: the average power consumption of block j per cycle,

—W^{rest^j}: the average power consumption of the processor per cycle excluding block j,

—C_O: the total number of cycles needed to execute an application when no prediction is made,

—C_P^j: the total amount of cycles needed when predictions are made for block j,

—A^j: total number of accesses to block j, and

—S^j: total number of accesses to block j saved by the predictor.

[3]Note that a 100% of saved accesses implies that the predictor predicts correctly all the useless accesses to the block *and* that there were no useful accesses to the block. If there are useful accesses, it is impossible to achieve 100% of saved accesses since these useful accesses eventually have to be done.

In the following, four assumptions are made for the sake of the simplicity of the analysis:

(1) An access to a block takes one processor cycle. This assumption is valid for the purpose of this work even if the block is fully pipelined (in this case, however, the latches between stages would be assumed to consume no power). To account for not fully pipelined blocks, an additional factor (i.e., the throughput of the block) should be considered in the analysis.

(2) The total number of accesses to a block, that is, A^j, is independent of whether the predictor exists or not. This may not be true since the presence of a predictor may increase the total amount of cycles needed to execute the application and, therefore, the number of accesses to the block can potentially be larger. Thus, for the analytical study to be accurate enough, the IPC degradation due to the predictor should be small.

(3) The power consumption of the predictor itself is negligible when compared to the power consumption of the block being predicted. This is true when simple predictors are used for power-hungry blocks, which is the case considered in this work. If the predictor for low power were complex, its power consumption should be considered in the following equations.

(4) When block j is disabled in a given cycle the power consumption of the processor per cycle is W^{rest^j}, independently of what the processor is doing in that cycle. Similarly, then the block is enabled, the power per cycle is $W^j + W^{\text{rest}^j}$.

Since the goal of a predictor is to reduce the energy required to execute a given application, it is needed that $E_P^j/E_O < 1$, E_O and E_P^j may be expressed as

$$E_O = C_O \times W^{\text{rest}^j} + A^j \times W^j$$

$$E_P^j = C_P^j \times W^{\text{rest}^j} + (A^j - S^j) \times W^j.$$

Let

$$\delta^j = \frac{W^j}{W^{\text{rest}^j}}.$$

Then

$$\frac{E_P^j}{E_O} = \frac{C_P^j + \delta^j \times (A^j - S^j)}{C_O + \delta^j \times A^j}.$$

When $E_P^j/E_O = 1$ no overall energy reduction is obtained. For this case,

$$\delta^j|_{\text{no red}} = \frac{C_P^j - C_O}{S^j}.$$

Since

$$\texttt{IPCRed}^{\texttt{j}} = \frac{\texttt{C}_\texttt{P}^{\texttt{j}} - \texttt{C}_\texttt{0}}{\texttt{C}_\texttt{P}^{\texttt{j}}} \; ; \; \texttt{SavAcc}^{\texttt{j}} = \frac{\texttt{S}^{\texttt{j}}}{\texttt{A}^{\texttt{j}}} \; ; \; \texttt{BlkAcc}^{\texttt{j}} = \frac{\texttt{A}^{\texttt{j}}}{\texttt{C}_\texttt{P}^{\texttt{j}}}$$

then $\delta^j|_{\text{no red}}$ can be rewritten as

$$\delta^j|_{\text{no red}} = \frac{\texttt{IPCRed}^j}{\texttt{SavAcc}^j \times \texttt{BlkAcc}^j}.$$

Moreover,

$$\texttt{WRatio}^j = \frac{W^j}{W^{all}} = \frac{W^j}{W^j + W^{\text{rest}^j}}.$$

Therefore, $\texttt{WRatio}^j|_{\text{no red}}$ is equal to

$$\frac{\delta^j|_{\text{no red}}}{\delta^j|_{\text{no red}} + 1} = \frac{\texttt{IPCRed}^j}{\texttt{IPCRed}^j + \texttt{SavAcc}^j \times \texttt{BlkAcc}^j}.$$

Thus, the predictor of block j will save energy if

$$\frac{E_P^j}{E_O} < 1 \Rightarrow \texttt{WRatio}^j > \frac{\texttt{IPCRed}^j}{\texttt{IPCRed}^j + \texttt{SavAcc}^j \times \texttt{BlkAcc}^j}.$$

Figure 4 plots $\texttt{WRatio}^j|_{\text{no red}}$ as a function of SavAcc, IPCRed, and BlkAcc. Given a fixed value of $\texttt{WRatio}^j$ (in the Figure 4 shown as w^j), the design space that presents overall processor power consumption savings is shown below the plane with equation $\texttt{WRatio}^j = w^j$. Figure 4 also serves the purpose of answering the question: given a predictor that renders SavAcc accesses saved with a penalty in performance of IPCRed on a particular block that is accessed in average BlkAcc times per cycle, how much power that block should account for the total processor power consumption for the technique to render a positive overall processor energy reduction.

Note that larger the $\texttt{BlkAcc}^j$, larger the design space with energy savings because more times the block will be disabled. In Figure 4, this design space corresponds to the curve comprised between the triangle shown on the base of each of the plots and the horizontal plane. As expected, the lower the IPC reduction, the smaller the SavAcc ratio can be and still be able to reduce the energy consumption of the design.

5.1 Extension to Two or More Predictors

As shown in Figure 2, more than one block of the processor may have a predictor associated. Let us assume that there are only two blocks with a predictor each. Two situations may happen:

—A predictor enables or disables its respective block independently of what the other predictor does.

—Some logic (shown in Figure 1(b) with the name of *predictor selector logic* (PSL)) may decide which of the two blocks to disable based on the predictions of both predictors and possibly on some other inputs from other blocks of the processor.

The first situation is a particular case of the second where the PSL just redirects the enable decisions from the predictors to the enable inputs of the

respective blocks. The second situation is more generic, and the PSL could implement different strategies:

—*disable at least one block.* If only one or both predictors want to disable, the PSL will do what the predictors decide. If none of the predictors wants to disable, the PSL should disable the block less sensitive to the IPC and/or more power hungry.
—*disable at most one block.* If both predictors want to disable, the PSL will have to decide the block that is not to disabled. If none or only one of the predictors want to disable, the PSL will do what the predictors decide.
—*disable none, one or both.* This strategy corresponds the first situation described before.

Each of these strategies has a different effect on the IPC and on the average and peak power-consumption reduction. Larger the number of blocks disabled, (a) potentially greater the IPC degradation, (b) larger the average power-consumption reduction and (c) larger the peak power-consumption reduction (however, in this case at least one of the blocks should be disabled every cycle to effectively reduce the peak power consumption).

A similar analytical model as the one plotted in Figure 4 may be obtained for the two-predictor scenario. The `IPCRed` parameter in this case is a function of the accuracy of both predictors and the strategy implemented by the PSL.

6. CASE EXAMPLE

In this section, a case example is evaluated. High-level estimations are presented to show that the methodology explained in the previous sections might produce significant power savings. The microprocessor example features two blocks with a predictor for low power associated with each one and the prediction selection logic. The two blocks are the on-chip level-two (L2) cache, when accessed for instruction fetches and the branch-target buffer (BTB), accessed for every branch instruction to obtain a prediction of its target address. Both the on-chip L2 cache and the BTB are large blocks that consume a significant amount of power. Reducing the amount of accesses to these blocks is expected to significantly reduce the overall power consumption. The purpose of this section is to show the benefits of the methodology of predicting a block result for low power. The ultimate overall microprocessor power reduction depends of several micro-architectural and circuit design parameters. In this work, only the amount of useless block accesses eliminated is evaluated.

There is one predictor associated with the L2 cache and two subpredictors to the BTB. The subpredictor selection logic will determine which of these two subpredictors will eventually make the prediction for the BTB.

In the following, the different predictors and both selection logics are explained in detail. Other more complex predictors may be devised. However, for the purpose of this work, simple predictors that provide a good accuracy versus design cost compromise will be described.

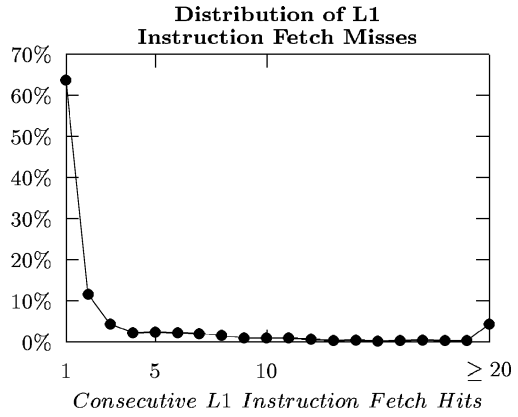


Fig. 5. Distribution of L1 instruction fetch misses as a function of consecutive L1 hit accesses. Workload is an average of the SPEC95 benchmarks.

6.1 On-Chip L2 Cache Predictor for Instruction Fetches (`WindowBasedPredictor`)

The proposed predictor for the L2 cache predicts the misses in the L1 instruction cache and access the L2 instruction cache concurrently with the L1 only in case a miss is predicted. On the other hand, if a hit is predicted, the L2 cache is only accessed if the prediction is incorrect; in this case the effective latency of the L2 cache is increased by the L1 latency of the hit/miss determination (considered in this work to be 1 cycle).

In this work, only the instruction fetches are predicted. The IPC degradation is around 3% when the L2 cache is accessed for instruction fetches only if there is an L1 cache miss with respect to the case in which both levels of cache are accessed in parallel. This percentage has been obtained using the same microarchitectural parameters used later on in the results section. Predicting the load accesses is also possible (see Section 4.1) but it has a larger impact on the IPC.

The rationale behind this predictor relies on the conjecture that the distance (in L1 cache accesses) between consecutive L1 misses is small. The author conveyed an experiment to prove the validity of this conjecture, and the results are shown in Figure 5, where, for a 2-way 16 KB L1 instruction cache, the percentage of the instruction fetches to the L1 cache that result in a miss is shown as a function of the number of consecutive L1 cache instruction fetch accesses that resulted in a hit. As an example, around 85% of all the misses occur within a distance of 5 or less. The shape of this distribution depends mainly on the workload and on the cache configuration.

The prediction algorithm (as already explained in Section 4.1, `Static Window-based`) defines a window size of W so that, after an instruction fetch miss occurs, the next miss is predicted to occur within the next W instruction fetch accesses to L1. If no miss occurs in these W accesses, a hit is predicted until the next miss occurs. By choosing the value of W, which is fixed once the predictor is implemented, the designer can select the fraction of L2 cache saved accesses versus the IPC reduction ratio; the larger (smaller) the W value,

smaller (larger) the fraction of L2 cache saved accesses and also smaller (larger) the IPC reduction.

6.2 BTB Predictor

A high accuracy in predicting the correct instruction stream is paramount in high-performance processors to achieve a competitive IPC metric [Perleberg and Smith 1993]. The penalty of a mispredicted branch instruction is several lost cycles, each of which could have derived one or more committed instructions. Therefore, a large BTB, usually organized as a cache, is necessary to provide as many correctly predicted target addresses as possible. The rather small 512-entry BTB of the Pentium Pro consumes about 5% of the processor power.

In our case study, the BTB, accessed for every branch instruction, has two subpredictors for low power and, therefore, a subpredictor selection logic (SSL) is needed to select which of the subpredictors will make the actual prediction. In the following, both subpredictors and the SSL are described.

Based on the Burstiness of BTB Misses (`BTBMissPredictor`*).* The first subpredictor relies on the conjecture that the misses in the BTB often happen in bursts, and correspond to control transfers to a new (or not recently executed) working zone of the code. The target addresses for the branches in the new working zone most likely will not be cached in the BTB. This conjecture was first described in Grunwald et al. [1998].

The subpredictor keeps track of how many consecutive BTB misses have occurred and it will predict that the next access to the BTB will be as well a miss in the case that number of consecutive misses is larger than a given number, henceforth named M, fixed by the designer. The larger M is the more consecutive BTB misses the subpredictor needs to detect in order to predict a miss in the next access to the BTB and, therefore, the smallest the fraction of BTB accesses are saved (but also the smallest the impact on the IPC). Thus, this predictor is again of the `Static Window-based` family described in Section 4.1.

Figure 6 shows how the BTB misses are distributed in bursts. For example, 50% of all the BTB misses happen in a single-miss burst, that is, there is at least one BTB hit right before and another one right after the BTB miss. In other words, 50% of the BTB misses are preceded by at least one BTB hit; for all these BTB misses, the predictor will mispredict them, since after a BTB hit the predictor described here always will predict hit.

Depending on the value of M, the number and type (hit or miss) of mispredictions varies. The predictor with smaller M has more mispredicted hits than the predictor with larger M; this implies that the branch prediction (and eventually the IPC) will be more adversely affected for the case with a small M. For this case, on the other hand, there are more BTB disables, thus saving more power.

Based on the Detection of Code Loops (`LoopPredictor`*).* The idea of this subpredictor is to detect the loops that the code may have, and store, in a special structure inside the predictor, the target addresses for the different

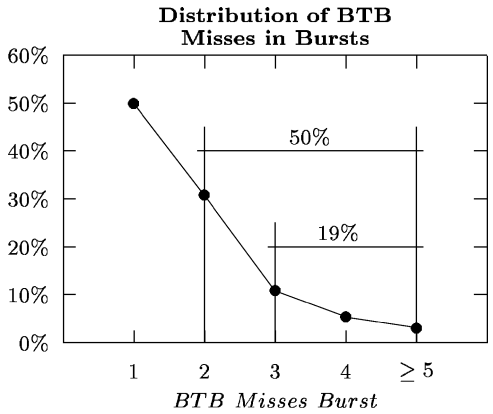


Fig. 6. Distribution of BTB misses in bursts for a typical BTB configuration. Workload is an average of the SPEC95 benchmarks.

branches within the loop. Thus, for nearly all the iterations of the loop, the target addresses of the branches within the loop may be provided by the predictor and not by the BTB (which is disabled during these iterations).

To reduce the cost of the predictor implementation, only some types of loops (those following the pattern of Figure 7(a)) will be handled. The simplest loop of this family of loops is depicted in Figure 7(b) (corresponds to $L = 1$ in Figure 7(a)), and it only consists of the mandatory closing branch. The hardware needed in the predictor to detect the loop is shown in Figure 7(b); the last branch address and its associated target address (provided by either the BTB or the predictor itself) are stored in two registers (`PastBranchAddress` (PBA) and `PastTargetAddress` (PTA) registers, respectively). Every time there is a BTB look-up, a comparator compares the current address of the branch to the value in the PBA register and, if equal, it implies that the simplest loop has been detected. Thus, for the next BTB look-ups till the end of the loop execution, the BTB may be disabled since the predicted target address will be provided by the PTA register. The end of the loop is detected when the comparison fails.

The same mechanism used to update the BTB with the correct target address when the branches are resolved by the execution units is applied to restore the PTA register in case its value is not correct.

The predictor for low power can be extended to handle more branches (always following the pattern in Figure 7(a)). For $L = 2$, the hardware needed is shown in Figure 7(d). After the predictor has provided its prediction, the *BTB look-up* signal clocks the current branch and target addresses into the PBA_2 and PTA_2 registers, respectively. The PTA and PBA registers behave as a FIFO. Thus, to detect an $L = 2$ loop, the oldest two branch addresses need to be the same as the newest two (including the current one).

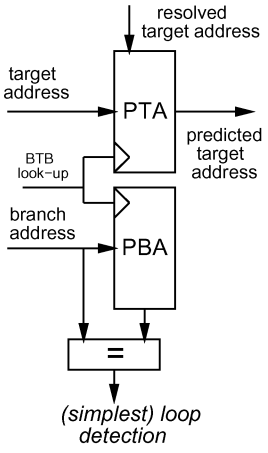
In general, to detect an L loop, $2L - 1$ PTA registers, $2L - 1$ PBA registers, L comparators, $L - 1$ multiplexers, and an L-input `AND` gate are needed. Figure 7(d) shows that the simplest loop detector comes almost for free (just an additional comparator). Therefore, the $L = 2$ predictor of the figure can detect

```
ini_loop:
        ...
        if cond_1 goto target_1
        ...
  target_1:
        ...
        if cond_2 goto target_2
        ...
  target_2:
        ...
        if cond_{L-1} goto target_{L-1}
        ...
  target_{L-1}:
        ...
        if not cond_{end_loop} goto ini_loop
end_loop:
        ...
```

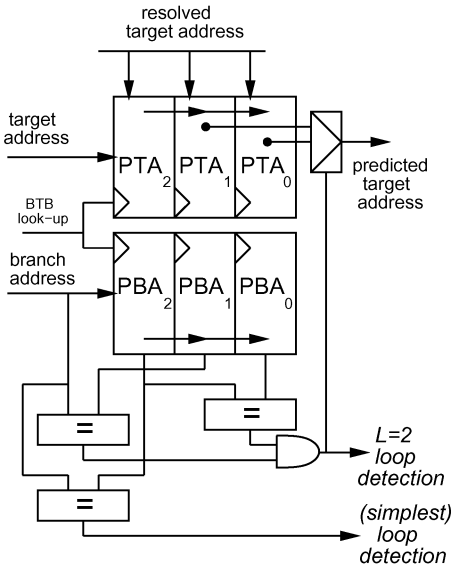
(a)

```
ini_loop:
        ...
        if not cond_{end_loop} goto ini_loop
end_loop:
        ...
```

(b)



(c)



(d)

Fig. 7. (a) Generic loop pattern detected by the predictor; (b) simplest loop; (c) hardware needed for $L = 1$; and (d) for $L = 2$.

both types of loops ($l = 1$ for the simplest predictor and for $l = 2$); in case both loop-detection signals get asserted, the signal corresponding to the loop with lower l has priority.

Figure 8 shows how many times (out of the total number of BTB accesses—look-ups and updates—when no predictor for low power is present) the predictor detects an l loop ($l = 1, \ldots, L$). The figure shows that, for an $L = 2$ predictor, 9.5% (4.8% due to $l = 1$ loops and 4.7% due to $l = 2$ loops) of the BTB accesses are saved.

Subpredictor Selection Logic. Since there are two subpredictors for low power for the BTB, some logic needs to arbitrate what subpredictor will make the prediction. The subpredictor selection logic (SSL) is described in Table V.

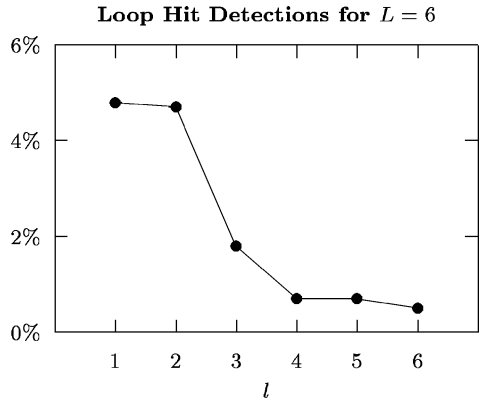


Fig. 8. Percentage of all BTB accesses (look-ups and updates) that result in loop-hit detections for $L = 6$. Workload is an average of the SPEC95 benchmarks.

Table V. Subpredictor Selection Logic

`BTBMiss Predictor` Predicts	`Loop Predictor` Predicts	SSL Action
Hit	No loop	Enables BTB
Miss	No loop	Disables BTB. Predicted target address is the fall-through path
— —	Loop	Disabled BTB. Predicted target address is provided by `LoopPredictor`

Two notes on the SSL behavior are

—it is static, that is, it does not maintain any history of the performance of the subpredictors to decide which one is better at any time. This static strategy keeps the complexity of the SSL low.

—when both subpredictors decide to disable the BTB (i.e., the `BTBMiss Predictor` predicts BTB miss, and the `LoopPredictor` detects a loop), the SSL decides that the predicted target address is provided by the `LoopPredictor` since this predictor is more accurate in providing the target address when it detects a loop hit than the `BTBMissPredictor` providing the fall-through path when predicts a miss (as it will be seen later on).

6.3 Prediction Selection Logic (PSL)

Similarly as with the subpredictor selection logic for the BTB subpredictors, some logic may regulate which of both blocks, the L2 cache and the BTB, are disabled based on the outcomes of their respective predictors. The three strategies explained in Section 5.1 will be evaluated in Section 6.4.

The only case in which the PSL has to override the outcome of one of the predictors is when it follows the *disable at most one block* strategy and both predictors want to disable their respective blocks. In this case, the PSL logic

will disable the L2 cache for instruction fetches, since it is less sensitive to the IPC than the BTB (as it will be seen later on).

6.4 Simulation Scenario

The SPEC95 benchmark suite will be used to evaluate the case example. The predictors and the PSL and SSL logic have been incorporated to the Simple Scalar v2.0 [Burger and Austin 1997] microarchitecture simulator.

The results reported are the weighted average for all the SPEC95 benchmarks. For each benchmark, 10M instructions are executed to warm up the blocks of the microprocessor, and during the following 100M instructions the statistics for the predictors for low power are gathered.

Several results will be provided in this section, each one representing a 3-value vector point in the IPC versus power-consumption design space. The values of the vector point are

—the percentage of IPC degradation (versus the IPC when no predictors for low power are present)
—the percentage of saved accesses to the BTB (out of all BTB accesses, including BTB updates)
—the percentage of saved accesses to the on-chip L2 cache (out of all instruction fetches)

The impact of the different PSL strategies will be investigated. The parameters for some of the predictors involved in the case example will be varied to see their effect. Finally, an estimation of the accuracy of the predictors for low power will be obtained as follows: one of the design points will be selected and its IPC will be compared against the IPC that it would have been obtained if the predictors had randomly behaved, following a uniform distribution, with the same percentage of saved accesses for each block as those of the selected design point.

The results presented are tightly coupled to the workload, and the underlying microarchitecture is simulated. For example, the L1 cache miss rate affects the amount of accesses to the L2 cache that may be saved. The branch predictor algorithm used in the simulations also affect the number of times the BTB is disabled. Nevertheless, the goal of this work is to show that the ideas described might produce a significant reduction in power consumed in a typical high-performance processor configuration.

6.4.1 *Results.* As explained above, there are three predictors for low power involved in the case example, and each has an associated parameter:

—for the on-chip L2 block (instruction fetches only):
  - `WindowBasedPredictor`: parameter W is the size of the window.
—for the BTB block:
  - `BTBMissPredictor`: parameter M is the number of consecutive BTB misses required to predict a miss in BTB.
  - `LoopPredictor`: parameter L is the number of branches of the loop.

Table VI. Effect on the IPC and Amount of Saved Accesses, for Different Parameter Values, of Each of the Predictors Involved in the Case Example

Block	Predictor	Parameters	Percentage of IPC Red.	Percentage of Saved Accesses
L2	`Window`	$W = 8$	0.5	76.5
		$W = 16$	0.3	71.0
		$W = 32$	0.2	64.7
BTB	`BTBMiss`	$M = 1$	3.6	5.7
		$M = 2$	1.0	1.8
		$M = 3$	0.5	0.8
	`Loop`	$L = 1$	0.7	4.8
		$L = 2$	2.4	9.5
		$L = 3$	3.0	11.3
	Both	$M = 1, L = 1$	4.3	10.5
		$M = 2, L = 2$	3.5	11.3
		$M = 3, L = 3$	3.5	12.1

Table VI shows the effect of the three predictors independently of each other, that is, when the predictors are applied individually, without any other predictor active (therefore, the PSL is not necessary). However, in the last row of the table both subpredictors for the BTB are active, and in this case the SSL strategy described in a Section 6.2 is applied (only three combinations of the M and L parameters are shown). For the case example and workload studied in this work, both the BTB and the L2 cache (for instruction accesses) are accessed about 0.5 times per cycle.

The following conclusions are derived from Table VI:

—The disabling of the on-chip L2 cache for instruction fetches by the `WindowBasedPredictor` degrades the least the IPC and obtains the largest percentage of saved accesses.[4]

—The `LoopPredictor` has less IPC degradation than the `BTBMissPredictor` at the same percentage of saved BTB accesses.[5]

—When both the `LoopPredictor` and the `BTBMissPredictor` are active

(a) the percentage of saved accesses is equal (or slightly less) than the addition of both individual percentages of saved accesses. The small difference accounts for those accesses to the BTB that where saved independently by both predictors and now are counted as saved only once (the SSL will give priority to the `LoopPredictor` for these accesses).

(b) the percentage of IPC degradation is also less or equal than the addition of both individual IPC degradations because the SSL gives priority to the more accurate `LoopPredictor` in case of conflict.

The PSL disabling strategies compared are (a) at least one of the blocks (henceforth named `1-or-2`, i.e., one or two blocks will be disabled all the time),

[4]Therefore, as explained in Section 6.3, the PSL, under the *disable at most one block* strategy, will disable the L2 cache when both the predictors for the L2 and the BTB want to disable their respective blocks.

[5]Therefore, as explained in Section 6.2, the SSL gives priority to the `LoopPredictor` over the `BTBMissPredictor` in case of conflict.

Table VII. PSL Strategy Results (W=16, M=2, L=2)

PSL Strategy	Percentage of IPC Degradation	Percentage of L2 Saved Accesses	Percentage of BTB Saved Accesses
`1-or-2`	7.6	98.3	11.3
`0-or-1`	1.1	70.0	1.4
`0-1-or-2`	3.7	70.0	11.3

(b) at most one block (henceforth named `0-or-1`), and (c) none, one or both (henceforth named `0-1-or-2`).

Table VII shows the effect of the PSL strategies for W=16, M=2, and L=2. This table indicates that

—the `0-or-1` PSL strategy degrades the least the IPC since it never allows the predictors for the L2 and the BTB to disable their respective blocks in the same cycle;

—the `1-or-2` strategy is the most aggressive in disabling the blocks; therefore, it has the largest IPC degradation and the largest power savings;

—the `0-1-or-2` strategy performs in between the previous two strategies.

Accuracy of the Predictors. The design points obtained in the previous section show the IPC reduction paid for a X% of L2 cache instruction fetch saved accesses and a Y% of BTB saved accesses. Clearly, the better the accuracy of the predictors for low power, more saved accesses, and less IPC reduction would have been obtained.

The following experiment has been run to obtain an estimation of how well the predictors have performed in terms of keeping the IPC degradation low: new "predictors" have been created for the L2 and BTB so that they will randomly disable the L2 and the BTB X% and Y% of the time, respectively. How this random predictor behavior affects the IPC?

The last design point in Table VII, that is, 70.0% of L2 saved accesses and 11.3% of BTB saved accesses at the cost of 3.7% IPC degradation, has been taken as an example. With the randomly behaved predictors, the IPC degradation dramatically raises to 10.7%, a penalty that certainly will slow down the efficiency of the microprocessor and will probably offset the power savings obtained on both blocks by the predictors. Therefore, the conclusion is that the simple predictors described in this work are reasonably accurate although better predictors can certainly be devised.

7. CONCLUSIONS

The power consumption of current and future high-performance microprocessors keeps increasing in spite of aggressive circuit design techniques and process shrinks. One of the reasons for this increase is the complexity of the microarchitecture required to achieve the IPC metric that each processor generation demands.

This work proposes a microarchitectural technique in which a prediction is made for some power-hungry blocks of a processor. The prediction consists of

whether the result of a particular block will be useful in order to execute the current instruction. If it is predicted useless, then that block is disabled and, therefore, no unnecessary power is consumed. Since the predictions are not totally accurate, the IPC might be degraded that in turn may offset the power savings obtained with the predictors.

A case example is presented where two power-hungry blocks have associated predictors for low power. The IPC versus power-consumption design space is explored for a particular microprocessor architecture, showing encouraging results.

REFERENCES

BANNON, P. 1998. Alpha EV7: A scalable single-chip SMP. In *Microprocessor Forum*.

BURD, T. AND PETERS, B. 1994. A power analysis of a microprocessor: The MIPS R3000 architecture. Tech. Rep., University of California at Berkeley.

BURGER, D. AND AUSTIN, T. 1997. The SimpleScalar tool set, version 2.0. Tech. Rep., Computer Sciences Department, University of Wisconsin-Madison.

CHANDRAKASAN, A., SHENG, S., AND BRODERSEN, R. 1992. Low power CMOS digital design. *IEEE Transactions SSC 27*, 4 (Apr.), 473–483.

DOBBERPUHL D. ET AL. 1992. A 200-MHz 64-b dual-issue CMOS microprocessor. *IEEE Journal on Solid State Circuits 27*, 11 (Nov.), 1555–1567.

GOWAN, M., BIRO, L., AND JACKSON, D. 1998. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th International Conference on Computer-Aided Design*, 726–731.

GRUNWALD, D., KLAUSER, A., MANNE, S., AND PLESZKUN, A. 1998. Confidence estimation for speculation control. In *ISCA'98*.

GWENNAP, L. 1998a. Power issues may limit future CPUs. *Microprocessor Report 10*, 10 (Aug.).

GWENNAP, L. 1998b. Shift to on-chip cache pays off. *Microprocessor Report 12*, 16 (Dec.).

JACOBSEN, E., ROTENBERG, E., AND SMITH, J. 1996. Assigning conficende to conditional branch predictions. In *Micro'96*.

JAGGAR, D. 1996. *Branch Cache*. Advanced Risc Machines Limited. US Patent no. 5506976.

KENNEDY, A. AND CROXTON, C. 1998. Pipelined processor operating in different power mode based on branch prediction state of branch history bit encoded as taken weakly, not taken and strongly not taken states. Motorola, Inc. US Patent no. 5740417.

KESSLER, R., MCLELLAN, E., AND WEBB, D. 1998. The Alpha 21264 microprocessor architecture. In *International Conference on Computer Design (ICCD'98)*, 90–95.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. 1997. The filter cache: An Energy efficient memory structure. In *International Symposium on Microarchitecture*.

MANNE, S., KLAUSER, A., AND GRUNWALD, D. 1998. Pipeline gating: speculation control for energy reduction. In *ISCA'98*.

MCFARLING, S. 1993. Combining branch predictors. Tech. Rep. WRL TN-36, Digital Western Research Laboratory.

MENG, T., GORDON, B., TSERN, E., AND HUNG, A. 1995. Portable video-on-demand in wireless communication. *Proc. IEEE 83*, 4 (Apr.), 659–680.

MUSOLL, E. 2000. Estimation of the upper-bound useless energy dissipation in a high-performance processor. In *Kool Chips Workshop (ISCA'00)*.

MUSOLL, E. AND LANG, T. 2002. Distance-based prediction of hit/miss of L1 caches. Tech. Rep., Department of Electrical and Computing Engineering, UC Irvine.

PERLEBERG, C. AND SMITH, A. 1993. Branch target buffer: Design and optimization. *IEEE Trans. Comp. 42*, 396–412.

SANTHANAM, S. 1996. StrongARM SA110, a 160 MHz 23 B 0.5 W CMOS ARM processor. In *Hot Chips VIII*.

SCOTT, J., LEE, L., ARENDS, J., AND MOYER, B. 1998. Designing the low-power M·Core™ architecture. In *Power Driven Microarchitecture Workshop*, 145–150.

TIWARI, V., SINGH, S., RAJGOPAL, S., MEHTA, G., PATEL, R., AND BAEZ, F. 1998. Reducing power in high-performance processors. In *Proceedings of the 35th International Conference on Computer-Aided Design*, 732–737.

YOAZ, A., EREZ, M., RONNEN, R., AND JOURDAN, S. 1998. Speculation techniques for improving load-related instruction scheduling. In *31st International Symp. on Microarchitecture*.

Received February 2002; revised July 2002; accepted February 2003