View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Hertfordshire Research Archive

Towards an Energy Efficient Branch Prediction Scheme Using Profiling, Adaptive Bias Measurement and Delay Region Scheduling

Michael Hicks, Colin Egan, Bruce Christianson, Patrick Quick
Compiler Technology and Computer Architecture Group (CTCA)
University of Hertfordshire, College Lane, Hatfield, AL10 9AB, UK
E-Mail: m.hicks@herts.ac.uk, c.egan@herts.ac.uk

***Abstract*—Dynamic branch predictors account for between 10% and 40% of a processor's dynamic power consumption. This power cost is proportional to the number of accesses made to that dynamic predictor during a program's execution. In this paper we propose the combined use of local delay region scheduling and profiling with an original adaptive branch bias measurement. The adaptive branch bias measurement takes note of the dynamic predictor's accuracy for a given branch and decides whether or not to assign a static prediction for that branch. The static prediction and local delay region scheduling information is represented as two hint bits in branch instructions. We show that, with the combined use of these two methods, the number of dynamic branch predictor accesses/updates can be reduced by up to 62%. The associated average power saving is very encouraging; for the example high-performance embedded architecture n average global processor power saving of 6.22% is achieved.**

Keywords: Branch Prediction, Power Consumption, Biased Branches, Profiling.

I. Introduction

The latency associated with branch instructions can be overcome by various means; these include branch prediction (dynamic and static), hardware multithreading, delayed branches and branch removal by aggressive static instruction scheduling. Currently, state-of-the-art processors tend to use dynamic branch prediction, but the use of dynamic predictors can consume large amounts of the silicon space and they can also consume large amounts of the power budget. In current processors, a branch predictor can consume between 10% and 40% of the overall CPU power budget [1]. The power cost is directly proportional to the number of accesses made to the dynamic predictor [2] and the power cost of modern dynamic branch predictors is comparable to that of a cache. In high performance architectures such costs may be acceptable, but we argue that such profligate use of silicon area is unlikely to be cost effective in low-power applications and will be an unnecessary drain on power. The effective use of silicon space and low power consumption is crucial for the embedded processor market. With the increasing pipeline depth of embedded processors, the accuracy of branch prediction is now becoming an important factor for embedded processor performance and a mispredicted branch will severely impact on performance. At the same time, power consumption must be kept to a minimum to ensure device usage longevity.

There is an equally valid counter argument that considers the power cost of a highly accurate dynamic predictor is offset by the effects of its accuracy [2]. Even though a dynamic predictor uses a great deal of power, the increased prediction accuracy and improved processor performance it provides results in power saving by the reduced number of branch mispredictions, negating the necessity of stalling the processor. This is because considerable power is consumed in a branch misprediction by the execution of instructions that cannot be allowed to be committed and recovery to some safe state.

In this paper, we present an approach for reducing the number of accesses and updates made to a dynamic branch predictor that combines scheduling the local delay region with profiling using an adaptive bias measurement. In the latter half of this paper, encouraging experimental results are presented. The results detail the extent to which the number of dynamic branch predictor accesses can be reduced, and also the amount of power that can be saved in an example architecture.

II. Predicting Biased Branches

In many cases the direction of a branch tends to be biased to either the taken path or to the not-taken path and therefore demonstrates a skewed distribution, which is alternatively referred to as bimodal. Using profiling, the frequency of executed branch paths can be determined and then used as a basis for predicting future runs of the program [4] [5]. In profile based prediction methodologies a static prediction-bit, or hint bit, is usually incorporated into the branch instruction format. This bit is used by the compiler to specify the prediction direction based on the branch's bias (taken or not-taken). Since they use the observed dynamic behaviour of branches, profile based branch prediction schemes differ from other static branch prediction schemes that use compile time heuristics. Also unlike dynamic branch prediction schemes, static profiling does not require large amounts of hardware support.

In this paper we use statically profiled branch prediction in conjunction with a dynamic predictor. The idea of statically profiled branch prediction is to avoid accessing the dynamic predictor whenever possible, thereby saving power. However, the profiled static prediction must be accurate to ensure that there is no impact on dynamic prediction accuracy, since inaccurate predictions are expensive in terms of both performance and power [6] [2] [7]. In the static code the number of biased branches appears to be small, but during program execution biased branches tend to be executed repeatedly and are therefore executed frequently [8]. Dynamic prediction is unnecessary for such biased branches as the direction of the branch can be statically profiled. This approach will reduce the number of accesses to the dynamic predictor and therefore save power.

Using some form of hint bits, a profiled static prediction can be used either to bypass the dynamic predictor or be used as a fallback when no prediction information is available dynamically. To save power in embedded processors it is desirable to remove dynamic predictions whenever possible. The drawback to removing biased branches from dynamic prediction using traditional compiler loop analysis is that any static prediction reflected by this method will almost always be less accurate than a dynamic prediction.

This drawback is intertwined with the definition of a biased branch. Previous approaches have used a fixed bias level [9], or, in effect, no particular bias level at all; a branch is simply marked as "likely to be taken" or "unlikely to be taken". Scant regard is given to how this will reconcile with the behaviour of the dynamic predictor in which it will be executing, and often the dynamic predictor will be more accurate [10]. Consequently, branch removal in this way impacts on performance and increases power consumption.

In our approach we take such problems into account and we propose the use of the dynamic bias measurement technique with profiling and local delay region scheduling. In local delay region scheduling the compiler schedules instructions from the same basic-block into the delay region following a branch instruction. These instructions, are those that would be executed irrespective of branch outcome and such that program semantics remain unaffected.

III. Adaptive Bias Measurement Through Profiling

Removing branches statically has traditionally been used either as an alternative to branch prediction, or, when used in conjunction with a dynamic predictor, it has been used as a fallback mechanism. The limitation of profiled static bias prediction from compiler branch heuristic analysis is accuracy. An improvement over simple heuristic static code analysis is to profile the compiled program at runtime to monitor how it behaves.

A. Profiling

Profiling, in this case, refers to the observation of a given program, at the assembly/machine level, while undergoing execution with a sample dataset [4] [5]. This means each branch instruction can be monitored in the form of a program trace by a detailed history of selected instructions and any relevant information extracted and used to form profiled static predictions where possible. A profiler is any application/system which can produce such data by observing a running program. The number of datasets that any given program is profiled with will affect the likely 'real' accuracy of the profiling results. Using a diverse range of datasets means the results will be more widely applicable.

The advantage of profiling over heuristic static code analysis is that a more precise boundary, or bias percentage, can be set for what constitutes a branch as heavily biased, and hence could be removed from dynamic prediction. Profiled traces permit the exact bias of a branch instruction to be known resulting in higher prediction accuracies.

B. Adaptive Bias Measurement

Profiling will be out performed by dynamic prediction for many branches unless the bias level is set so high that only extremely biased branches are removed and therefore profiling should be used with due caution. Consequently, we only only assign a profiled prediction to a branch where avoiding dynamic prediction has no significant negative impact on that branch's dynamic prediction accuracy.

When profiling each branch in a program's execution, an ideal profiler records the directional history for each branch, and also the prediction history [10] [5]. From this record or trace, we compute whether a branch's bias is equal to, or greater than its associated prediction accuracy from the dynamic predictor. This computation is key to the results we present in this paper. Assuming a program is profiled against an adequately varied data set, we show that these branches can safely be removed from dynamic prediction through the use of profiled hint bits. This approach to bias measurement also has a beneficial side-effect that it removes a significant number of difficult-to-predict branches. Furthermore, a branch with a very low prediction accuracy, but a higher bias will be caught by this method. Difficult-to-predict branches have a significant impact on both performance and power consumption [6] [8].

IV. Local Delay Region

Local delay region scheduling is the process of scheduling branch independent instructions from before the branch in the same basic-block into the delay slots to be executed by the processor after the branch. A branch independent instruction is any instruction whose result is not directly, or indirectly, depended upon by the branch to compute its own behaviour. The locally scheduled delay region, for a given branch, is executed irrespective of the branch direction outcome, and removes the need to predict for any branch where it can be used. Figure 1 demonstrates the process.

It is not beneficial to use local delay region scheduling in well optimised code and, where the delay region is very large such as in deeply pipelined processors. This study is focused on the embedded market where the number of delay slots tend

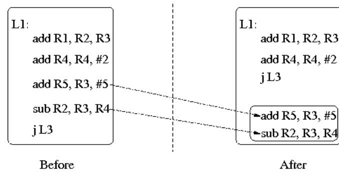

Fig. 1. Local delayed branch scheduling

to be low. Local delay region scheduling can be useful by careful use and leaving other branches untouched for dynamic prediction. Local delay region scheduling works particularly well for an unconditional absolute branch that has a fixed target address. Consequently, we propose the use of the local delay region in conjunction with adaptive bias measurement.

V. H/W IMPLEMENTATION

The hardware modifications required to convey information about static predictions, and therefore avoid the dynamic predictor for a given branch, are relatively simple.

Many modern processors already predecode instructions to determine whether to access the dynamic predictor unit; in which case, hint information need only be included with the branch instructions themselves. Some modern embedded instruction sets [11] already include hint bits in branch instructions, although they are only used as a fallback. We incorporate two hint bits into the branch instruction format, where the two hint bits provide profiled branch behaviour information. To our knowledge no compiler makes use of the two hint bits as we describe in this paper, which represent the following branch behaviour:

1) Statically predict taken. Do not access, or update, the dynamic predictor for this branch.
2) Statically predict not-taken. Do not access, or update, the dynamic predictor for this branch.
3) Use the locally scheduled delay region. Do not access, or update, the dynamic predictor for this branch.
4) Use the dynamic predictor.

By default all instructions would be set to case 4. The algorithm that describes how these hint bits are set is explained in the following section.

In case 1, we have hinted that the branch should always be assumed taken. This means that no access is required to the direction prediction logic in the branch predictor unit. However, the processor does not know until the decode stage what the target address will be. Rather than any complexities of computation, this is largely down to there being several different formats for branch instructions, and thus the position of the target bits in the instruction is not known. Our simulations have shown us that this is not the problem it seems at first: over 75% of dynamic branches found to be dynamically biased fall into a single instruction format - typically a single kind of offset-branch (another 18% are absolute jump instructions, but for these we use local delay region scheduling). This means that with minimal hardware modification, simple logic can be introduced to produce the target address for case 1 branches, and hence avoid accessing the Branch Target-address Cache (BTC) for predicted taken branches; this is significant for power aware designs. In the case that the hint bits provide an incorrect prediction, the existing dynamic branch prediction logic is used to recover in the same way as a dynamic misprediction (case 4).

The hardware modifications are shown in Figure 2. The block below the I-Cache represents a fetched example instruction (in this case a hinted taken branch). The simple decoder is a very small piece of hardware required to decode the two hint bits from an instruction into the relevant processor signals. The label for the local delay region is simply to indicate that this hint must be used later (in the exe pipeline stage). Additionally, outside of this diagram, during execution pipeline stage, the two hint bits must also be examined to ensure that no update occurs to the branch predictor which is also very important for power aware processors.

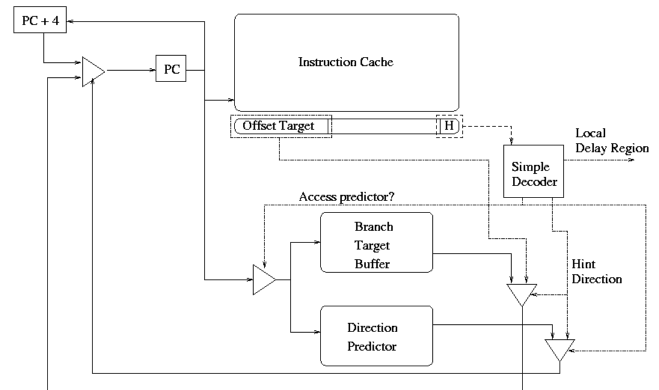

Fig. 2. A simplified block diagram representation of the hardware modifications required in the Instruction Fetch stage for the hardware simplicity approach, in order to implement the universal hint bits

VI. SIMULATIONS

This section details the process of implementation and testing of our dynamic branch prediction reduction methods, and the associated hardware modifications.

A. EEMBC and Wattch

We use the Electronic Embedded Microprocessor Benchmark Consortium (EEMBC) benchmarks [12]. EEMBC was chosen instead of the SPEC benchmarks as they represent a more appropriate target for this type of algorithm.

EEMBC [12] is a benchmark suite consisting of around forty separate benchmarks that are divided into five sections, or subsuites: Automotive, Consumer, Networking, Office and Telecom. Each subsuite represents a further specialisation towards a particular behaviour characterisation. Table I shows a simple breakdown of the five subsuites in EEMBC. Every benchmark in the suite was executed to completion to generate the results shown in the next subsection.

We use a modified version of Wattch [13], which itself is a variation of the SimpleScalar processor simulator. Wattch uses the Portable Instruction Set [Architecture] (PISA), which

TABLE I
THE FIVE EEMBC SUBSUITES WITH A LIST OF SOME OF THE TYPES OF BENCHMARKS CONTAINED WITHIN EACH SUITE

SubSuite	Benchmarks (selection of largest)
Automotive	Angle-To-Time Conversion, Fast Fourier Transform, Matrix Math...
Consumer	JPEG Compression/Decompression, RGB to CMYK, Grayscale image filter...
Networking	IP Reassembly, Network Address Translation, Route Lookup...
Office	Bezier Curve Interpolation, Floyd-Stein Grayscale Dithering, Bitmap Rotation...
Telecom	Autocorrelation, Convolutional Encoder, Viterbi Decoder...

is is a variant of the Mips instruction set. The Wattch pipeline has seven stages, and two branch delay slots before branch resolution. Local delay region scheduling was used to remove dynamic predictor accesses for the unconditional absolute-jump instruction formats, and our profiled adaptive bias measurement was used to remove dynamic predictor accesses for appropriately biased offset branch format instructions. The static prediction/delay region usage information was represented using two redundant bits in the branch instructions of the PISA instruction set. The required simulated hardware modifications were minimal and were configured as described in the previous section.

B. Algorithm

The algorithm used to configure the two hint bits in each branch instruction was implemented as shown in Algorithm 1.

Input: All Assembly Files of Programs
Output: Appropriately Hinted Assembly Files
foreach *Program* **do**
  foreach *Assembly File* **do**
    foreach *Branch Instruction* **do**
      Initially, set hint bits to "Use the dynamic predictor for this branch"
      if *Branch == Unconditional Branch* **then**
        Set hint bits to use local delay region and move two instructions preceding branch into delay region (if possible)
      else
        if *Branch's Profiled Bias ≥ Dynamic Branch Predictor's Accuracy for this Branch* **then**
          Set hint bits to Predict Profiled Bias
        end
      end
    end
  end
end

Algorithm 1: Dynamic Branch Prediction Reduction Algorithm

Figure 4 3 shows our profiling and hinting mechanism. All of the runtime profiling was conducted on a 'training' input dataset. The results shown in the next subsection were produced using a different 'test' dataset to ensure that no bias to a particular dataset was represented.

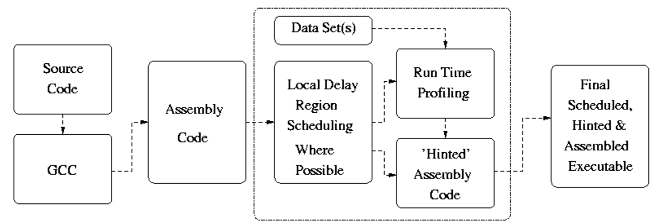

Fig. 3. Block model of the profiling and hinting regime

C. Simulation Results

The results presented and discussed in this section were produced using the processor configuration shown in Table II. Since the first part of these results is primarily observing branch instruction accuracy, the most important variable to note is branch predictor used. The GShare predictor [14] was chosen for it's accuracy, size and general applicability. The other specifications of the system were selected as a representation of a modest high-performance embedded CPU for use in applications such as PDAs and mobile telephones.

TABLE II
BASELINE CONFIGURATION USED TO GENERATE THE RESULTS SHOWN IN THIS SECTION

HWattch Parameter	Value
Issue/Decode Width	2 Instructions
Delay Slots	3
Branch Predictor	GShare
Direction Predictor Table Entries	1024
BP History Width	8Bits
BTB Sets/Associativity	512 Entries/4 Way
Data Cache Size	1024 Sets/64bit Block Size/4Way
Instruction Cache Size	512 Sets/32bit Block Size
Clock Gating Regime Modelled	Aggressive conditional clocking (non-ideal) 15% power dissipation with zero accesses
Compiler for EEMBC	gcc -O2

Table III shows the occurrence of different branch instructions across the execution of the entire EEMBC benchmark suite. The static occurrence refers to the proportion of branches accounted for by a particular branch type in the static assembly code. Dynamic occurrence refers to the proportion of branches accounted for dynamically by a particular branch type. Additionally, the third column shows whether this branch type can be removed from dynamic prediction by either of the techniques proposed. From this table we can see that the majority of dynamic branches are potential candidates for removal by either technique. Why a particular method can be used, or not, is explained in the previous section.

1) Dynamic Predictor Access Reduction: Table IV shows the success of using local delay scheduling and adaptive bias measurement to remove accesses to the branch predictor. All values shown in Table IV are averages for each subsuite. Averages were used due to the vast number of benchmarks in each suite, and also because of the high behavioural similarities of

TABLE III
STATIC AND DYNAMIC BRANCH OCCURRENCE FOR EACH PISA BRANCH TYPE, AND WHICH DYNAMIC ACCESS REMOVAL METHOD CAN BE USED

Branch Instruction	Static Occurrence	Dynamic Occurrence	Applicable Method
j	10.21%	17.31%	Local Delay Region
jal	33.95%	3.58%	Local Delay Region
jr	15.54%	3.55%	None Used
jalr	2.32%	0.04%	None Used
beq	18.18%	20.23%	Bias Profiling
bne	16.46%	50.09%	Bias Profiling
blez	1.52%	2.58%	Bias Profiling
bgtz	0.27%	1.04%	Bias Profiling
bltz	0.48%	0.39%	Bias Profiling
bgez	1.06%	1.19%	Bias Profiling

each suite. The average results were calculated by taking the total across a whole subsuite and using it as the divisor for the sum of the measured variable across the whole subsuite. For instance, 'Static Hint Rate' was calculated by summing all statically hinted branches across an entire subsuite, and dividing by the entire subsuite sum of branches – not by unweighted averaging. Figure 4 represents the same results in graphical form with the addition of an overall average.

TABLE IV
RESULTS OF LOCAL DELAY REGION AND ADAPTIVE BIAS HINTING FOR EACH EEMBC SUBSUITE

SubSuite	Dynamic Branch Rate	Static Hint Rate	Dyamic Access Reduction	Dynamic Stream Change
Automotive	17.57%	22.10%	56.42%	0.92%
Consumer	17.22%	30.80%	63.53%	0.49%
Networking	24.56%	10.57%	62.50%	0.48%
Office	19.57%	30.76%	46.82%	0.1%
Telecom	12.37%	20.49%	51.71%	-0.03%
Average	18.59%	18.29%	62.01%	0.48%

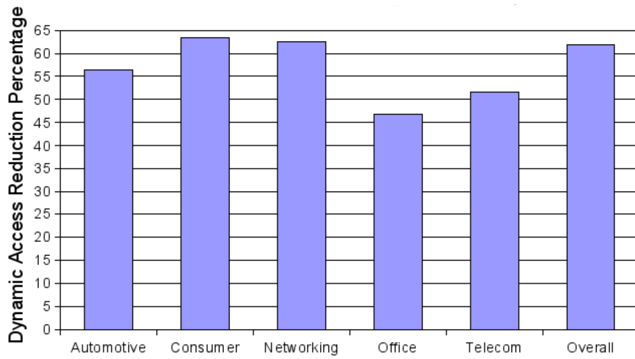

Fig. 4. The average percentage access reduction to the dynamic branch predictor for each EEMBC Subsuite

Each column in Table IV demonstrates the following:

- Dynamic Branch Rate – The percentage of instructions in the dynamic stream that were branch instructions
- Static Hint Rate – The percentage of static branches that were hinted
- Dynamic Access Reduction – The resulting reduction in accesses to the dynamic branch predictor (both the direction predictor and BTB)
- Dynamic Stream Change – The change in size (number of instructions executed) of the dynamic instruction stream as a result of the static hinting

These results demonstrate the effectiveness of the combination of local delay region scheduling and adaptive bias measurement for removing the need to dynamically predict for many branches. The overall average of 62% dynamic branch prediction reduction is extremely promising. Unsurprisingly, the Consumer subsuite performed most successfully with the algorithm. This is because of the highly cyclic nature of many of the algorithms included in this subsuite: JPEG compression and decompression for instance.

Most importantly, we can see that our dynamic branch prediction reduction algorithm has no significant detrimental effects on the performance of the program: Table IV shows that the size of the dynamic instruction stream was not significantly expanded with additional instructions from increased missprediction. In fact, in many individual cases, the number of instructions executed was reduced. This is likely accounted for by the removal of poorly dynamically predicted branches; a branch with a bias greater than its accuracy is automatically removed from dynamic prediction.

Although the average results in Table IV are representative, there were some intrasubsuite exceptions. Notably these were: Angle-To-Time Conversion benchmark in Automotive and the Viterbi Decoder benchmark in Telecom. These had dynamic prediction removal percentages of 11% and 35%, respectively.

2) Subsequent Power Saving: After simulating the number of dynamic branch predictor accesses that could be removed for the predictor used in our example architecture we then used our variant of Wattch to model the amount of power than can be saved. The dynamic branch predictor accounts for between 10% and 15% of global power dissipation in the example architecture when all branches use dynamic prediction.

Demonstrating how much power can be saved is indicative only for the example architecture used. The amount of power saved in the branch predictor itself is generally proportional to the dynamic access reduction as a result of the application of our algorithm. However, the power saved in the branch predictor gives no indication of the global power saved over the whole processor, and also does not take into account any additional delay incurred by the use of the access reduction algorithm. Providing global processor power results is thus useful, but the results depend on the relative size of the rest of the processor compared to the dynamic branch predictor unit.

The average global processor power savings per committed instruction, for the architecture in Table II, are shown in Table V. The results per committed instruction were calculated by dividing the total global power consumed during a program's execution by the number of committed instructions. The power saving per committed instruction implicitly takes into account any change in the size/delay of the instruction stream as the number of committed instructions remains the

same for all test executions of the benchmarks; an increase of the number of instructions executed after the application of the reduction algorithm would scale the power saved in the branch predictor when calculated for the committed instructions even though branch predictor accesses may have been reduced.

TABLE V
AVERAGE POWER SAVING PER COMMITTED INSTRUCTION FOR NON-IDEAL CLOCK GATING REGIME AND *IDEAL CLOCK GATING REGIME

EEMBC Subsuite	Average Power Saving	Best/Worst
Automotive	5.43% (*12.38%)	14.87% / 10.15%
Consumer	6.17% (*12.69%)	10.66% / 9.73%
Networking	6.84% (*14.53%)	14.73% / 10.60%
Office	5.66% (*13.56%)	11.38% / 10.07%
Telecom	4.10% (*10.07%)	11.14% / 9.21%
Overall	6.22% (*13.47%)	N/A

The standard power saving results in Table V are for the non-ideal clock gating regime described in the processor specification Table II. However, we have additionally included the power saving results for a more ideal clock gating algorithm with close to zero dissipation on zero accesses. These additional results are denoted with an asterisk (*).

Figure 5 shows the standard, non-ideal results, but also includes the average power saving per instruction as if no accesses were made to the dynamic branch predictor (Free BP – whilst still maintaining the same prediction accuracy). This allows a comparison between the success of the power saving and the absolute ceiling value possible.

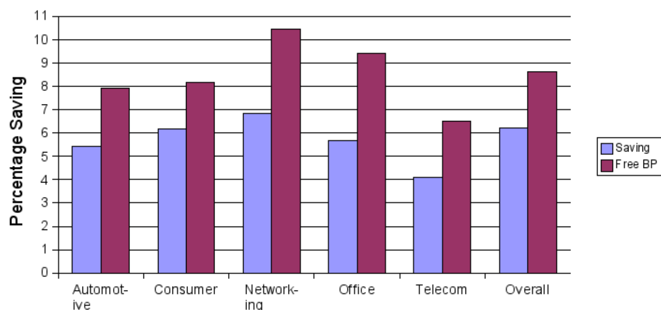

Fig. 5. Average power saving per committed instruction

Although Figure 5 shows that the algorithm is not 'perfect', we must remember that not all branch prediction accesses are removed and as such it will not be possible to be ideal without impacting heavily on performance, and thus power.

VII. CONCLUSIONS AND FUTURE WORK

Dynamic branch predictors cannot be removed from processors while high performance and low power consumption are issues [6]. However, the results in this paper have shown that, in an embedded context, the number of accesses that need to be made throughout a program's execution can be dramatically reduced: in this example architecture by 62%. While previous attempts at power saving have focused on the introduction of hardware units to monitor dynamic behaviour, our approach can achieve similar levels of access reduction, but without the need to significantly modify hardware. The number of dynamic predictor accesses that can be removed with this approach is highly dependent on the accuracy of the dynamic predictor being used. In this paper we used a very accurate dynamic predictor, but a higher reduction can be achieved in architectures with a less accurate dynamic branch predictor. Additionally, this approach is applicable to both SuperScalar and VLIW processors.

The amount of power saved, for the non-ideal clock gating regime, averaged at 6.22% global power saving across the EEMBC benchmark suites. This result is significant and very encouraging; in architectures where the branch predictor is relatively more expensive (in terms of power) this figure will be higher. When considering the predecode logic that already exists in most processors, the hardware modifications are minor and easy to accommodate. The time required to simulate, profile, assign static predictions and generate results was under 90 minutes for the entire EEMBC suite on a standard modern desktop machine, and this process is required only once before a program's distribution. For these small costs, an average power saving of at least 6% is highly attractive for processors in embedded devices that account for a large proportion of the whole device's limited energy budget.

REFERENCES

[1] Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.R.: Power issues related to branch prediction, IEEE HPCA (2002)
[2] Egan, C., Hicks, M., Christianson, B., Quick, P.: Enhancing the i-cache to reduce the power consumption of dynamic branch predictors, IEEE Digital System Design (July 2005)
[3] Seng, J.S., Tullsen, D.M.: Exploring the potential of architecture-level power optimizations, PACS (2003)
[4] Fisher, J.A., Freudenberger, S.: Predicting conditional branch directions from previous runs of a program. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston. (October 1992)
[5] Hicks, M., Egan, C., Christianson, B., Quick, P.: Htracer: A dynamic instruction stream research tool, IEEE Digital System Design (July 2005)
[6] Parikh, D., Skadron, K., Zhang, Y., Stan, M.: Power aware branch prediction: Characterization and design. IEEE Transactions On Computers **53**(2) (February 2004)
[7] Martin, A.J., Nystrom, M., Penzes, P.L.: Et^2: A metric for time and energy efficiency of computation. (2003)
[8] Vintan, L., Gellert, A., Florea, A., Oancea, M., Egan, C.: Understanding prediction limits through unbiased branches. In: Advances In Computer Systems Architecture. Volume 4186-0480 of Lecture Notes In Computer Science., Springer-Verlag (September 2006) 480487
[9] Jacobsen, E., Rotenberg, E., Smith, J.: Assigning confidence to conditional branch predictions, IEEE 29th International Symposium on Microarchitecture (1996)
[10] Hicks, M., Egan, C., Christianson, B., Quick, P.: Reducing the branch power cost in embedded processors through static scheduling, profiling and superblock formation. In: Advances In Computer Systems Architecture. (September 2006)
[11] IBM: PowerPC Instruction Set Manual. (2005)
[12] Levy, M.: The embedded microprocessor benchmark consortium. Online (2005)
[13] Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations, 27th annual international symposium on Computer architecture (2000)
[14] Egan, C.: Dynamic Branch Prediction In High Performance Super Scalar Processors. PhD thesis, University of Hertfordshire (August 2000)