Reducing the Branch Power Cost In Embedded Processors Through Static Scheduling, Profiling and SuperBlock Formation

Michael Hicks, Colin Egan, Bruce Christianson, Patrick Quick

Compiler Technology and Computer Architecture Group (CTCA)
University of Hertfordshire, College Lane, Hatfield, AL10 9AB, UK
m.hicks@herts.ac.uk

Abstract. Dynamic branch predictor logic alone accounts for approximately 10% of total processor power dissipation. Recent research indicates that the power cost of a large dynamic branch predictor is offset by the power savings created by its increased accuracy. We describe a method of reducing dynamic predictor power dissipation without degrading prediction accuracy by using a combination of local delay region scheduling and run time profiling of branches. Feedback into the static code is achieved with hint bits and avoids the need for dynamic prediction for some individual branches. This method requires only minimal hardware modifications and coexists with a dynamic predictor.

1 Introduction

Accurate branch prediction is extremely important in modern pipelined and MII microprocessors [10] [2]. Branch prediction reduces the amount of time spent executing a program by forecasting the likely direction of branch assembly instructions. Mispredicting a branch direction wastes both time and power, by executing instructions in the pipeline which will not be committed. Research [8] [3] has shown that, even with their increased power cost, modern larger predictors actually save global power by the effects of their increased accuracy. This means that any attempt to reduce the power consumption of a dynamic predictor must not come at the cost of decreased accuracy; a holistic attitude to processor power consumption must be employed [7][9].

In this paper we explore the use of delay region scheduling, branch profiling and hint bits (in conjunction with a dynamic predictor) in order to reduce the branch power cost for mobile devices, without reducing accuracy.

2 Branch Delay Region Scheduling

The branch delay region is the period of processor cycles proceeding a branch instruction in the processor pipeline before branch resolution occurs. Instructions can fill this gap either speculatively, using branch prediction, or by the use of scheduling. The examples in this section use a 5 stage MIPS pipeline with 2 delay slots.

2.1 Local Delayed Branch

In contrast to scheduling into the delay region from a target/fallthrough path of a branch, a locally scheduled delay region consists of branch independent instructions that precede the branch (see Figure 1). A branch independent instruction is any instruction whose result is not directly or indirectly depended upon by the branch to calculate its own behaviour.

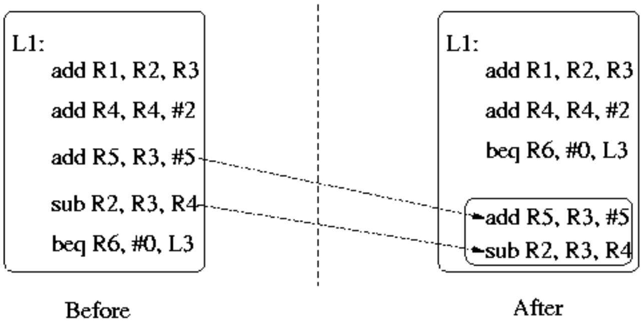

Fig. 1. An example of local delayed branch scheduling.

Deciding which instructions can be moved into the delay region locally is straightforward. Starting with the instruction from the bottom of the given basic block in the static stream, above the branch, examine the target register operand. If this target register is NOT used as an operand in the computation of the branch instruction then it can be safely moved into the delay region. This process continues with the next instruction up from the branch in the static stream, with the difference that this time the scheduler must decide whether the target of the instruction is used by any of the other instructions below it (which are in turn used to compute the branch).

Local Delay Region Scheduling is an excellent method for utilising the delay region where possible; it is always a win and completely avoids the use of a branch predictor for the given branch. The clear disadvantage with local delay region scheduling is that it cannot always be used. There are two situations that result in this: well optimised code and deeply pipelined processors (where the delay region is very large). It is our position that, as part of the combined approach described in this paper, the local delay region is profitable.

3 Profiling

Suppose that we wish to associate a reliable static prediction with as many branches as possible, so as to reduce accesses to the dynamic branch predictor of a processor at runtime (in order to save power). This can be achieved to a reasonable degree through static analysis of the assembly code of a program; it is often clear that branches in loops will commonly be taken and internal break points not-taken.

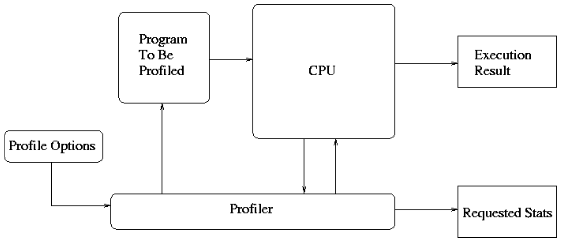

Fig. 2. The profiler is supplied with parameters for the program and the traces/statistics to be logged

A more reliable method is to observe the behaviour of a given program while undergoing execution with a sample dataset [4]. Each branch instruction can be monitored in the form of a program trace and any relevant information extracted and used to form static predictions where possible. A profiler is any application/system which can produce such data by observing a running program (see Figure 2). The proceeding two sections examine the possibility of removing certain classes of branch from dynamic prediction by the use of run-time profiling.

3.1 Biased Branches

One class of branches that can be removed from dynamic prediction, without impacting on accuracy, are highly biased branches. A biased branch is a branch which is commonly taken or not taken, many times in succession before possibly changing direction briefly. The branch has a bias to one behaviour. These kinds of branches can, in many cases, be seen to waste energy in the predictor since their predicted behaviour will be almost constantly the same [5] [8].

The principles of spatial and temporal locality intuitively tell us that biased branches account for a large proportion of the dynamic instruction stream. Identifying these branches in the static code and flagging them with an accurate static prediction would enable them to be executed without accessing the dynamic predictor. The profiler needs to read the static assembly code and log, for each each branch instruction during profiling, whether it was taken or not taken at each occurrence.

3.2 Difficult to Predict Branches (Anti Prediction)

Another class of branch instructions that would be useful to remove from dynamic branch predictor accesses are difficult to predict branches. In any static program there are branches which are difficult to predict and which are inherently data driven. When a prediction for a given branch is nearly always likely to be wrong, there is little point in consuming power to produce a prediction for it since a number of stalls will likely be incurred anyway [5] [8] [6].

Using profiling, it is possible to locate these branches at runtime using different data sets and by monitoring every branch. The accuracy of each dynamic prediction is required rather than just a given branch's behaviour. For every branch, the profiler needs to compare the predicted behaviour of the branch with the actual behaviour. In the case of those branch instructions where accuracy of the dynamic predictor is consistently poor, it is beneficial to flag the static branch as difficult to predict and avoid accessing the branch predictor at all, letting the processor assume the fallthrough path. Accordingly, filling the delay region with NOP instructions wastes significantly less power executing instructions that are unlikely to be committed.

4 Combined Approach Using Hint Bits

The main goal of the profiling techniques discussed previously can only be realised if there is a way of storing the results in the static code of a program, which can then be used dynamically by the processor to avoid accessing the branch prediction hardware [3].

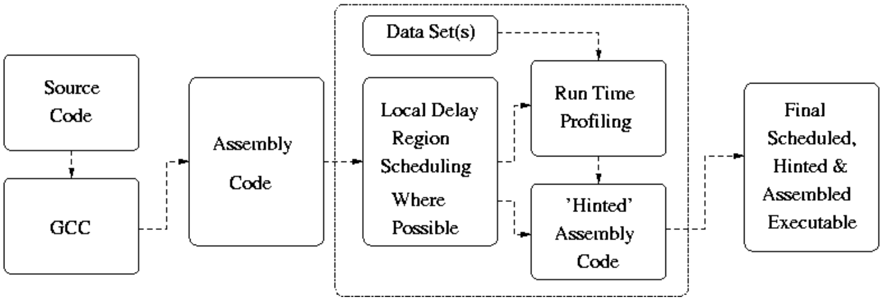
Source Code
GCC
Assembly Code
Data Set(s)
Local Delay Region Scheduling Where Possible
Run Time Profiling
'Hinted' Assembly Code
Final Scheduled, Hinted & Assembled Executable

Fig. 3. Block diagram of the proposed scheduling and hinting algorithm. The dotted box indicates the new stages introduced by the algorithm into the creation of an executable program

The combined approach works as follows:

1. Compile the program, using GCC for instance, into assembly code.
2. The Scheduler parses the assembly code and decides for which branch instructions the local delay region can be used (see section 2.1).
3. The Profiler assembles a temporary version of the program and executes it using the specified data set(s). The behaviour of each branch instruction is logged (see section 3).
4. The output from the profiling stage is used to annotate the delay scheduled assembly code.
5. Finally, the resulting annotated assembly code is compiled and linked to form the new executable.

The exact number of branches that can be eliminated from runtime predictor access in the target program depends upon the tuning of the profiler and the number of branches where the local delay region can be used.

4.1 Hint Bits

So far we have described a process of annotating branch instructions in the static assembly code to reflect the use of the local delay region and of the profiling results. The way this is represented in the assembly/machine code is by using an existing method known as hint bits (though now with the new function of power saving).

The four mutually exclusive behaviour hints in our algorithm which need to be stored are:

1. Access the branch predictor for this instruction.
2. or Assume this branch is taken (don't access dynamic predictor logic).
3. or Assume this branch is not taken (don't access dynamic predictor logic).
4. or Use this branch's local delay region (don't access dynamic predictor logic).

The implementation of this method requires two additional bits in an instruction. Whether these bits are located in all of the instruction set or just branches is discussed in the proceeding section. Another salient point is that the information in a statically predicted taken branch replaces only the dynamic direction predictor in full; the target of the assumed taken branch is still required. Accessing the Branch Target Buffer is costly, in terms of power, and must be avoided.

Most embedded architectures are Reduced Instruction Set Computers [8]. Part of the benefit of this is the simplicity of the instruction format. Since most embedded system are executing relatively small programs, many of the frequently iterating loops (the highly biased branches, covered by the case 2 hint) will be PC relative branches. This means that the target address for a majority of these branches will be contained within a fixed position inside the format. This does not require that the instruction undergo any complex predecoding, only that it is offset from the current PC value to provide the target address. Branch instructions that have been marked by the profiler as having a heavy bias towards a taken path, but which do not fall into the PC relative fixed target position category have to be ignored and left for dynamic prediction.

The general 'hinting' algorithm:

1. Initially, set the hint bits of all instructions to: assume not taken (and do not access predictor).
2. Set hint bits to reflect use of the local delay region where the scheduler has used this method.
3. From profiling results, set hint bits to reflect taken biased branches where possible.
4. All remaining branch instructions have their hint bits set to use the dynamic predictor.

4.2 Hardware Requirements/Modifications

The two possible implementation strategies are:

Hardware Simplicity: Annotate every instruction with two hint bits. This is easy to implement in hardware and introduces little additional control logic. All non branch instructions will also be eliminated from branch predictor accesses. The disadvantages of this method are that it requires that the processor's clock frequency is low enough to permit an I-Cache access and branch predictor access in series in one cycle and that there are enough redundant bits in all instructions.

Hardware Complexity: Annotate only branch instructions with hint bits and use a hardware mechanism similar to a Prediction Probe Detector [8] to interpret hint bits. This has minimal effect on the instruction set. It also means there is no restriction to series access of the I-Cache then branch predictor. The main disadvantage is the newly introduced PPD and the need for instructions to pass through the pipeline once before the PPD will restrict predictor access.

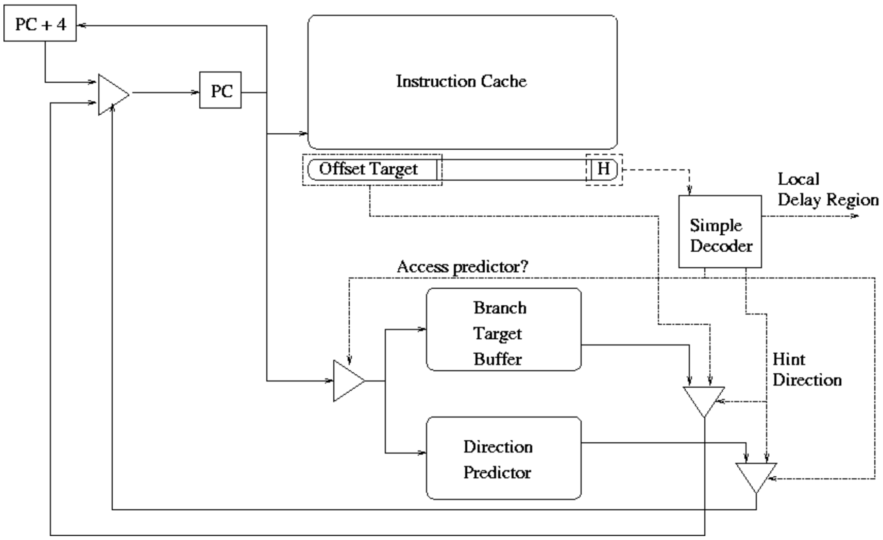

Fig. 4. Diagram of required hardware modifications. The block below the I-Cache represents a fetched example instruction (in this case a hinted taken branch).

The hardware simplicity model offers the greatest power savings and is particularly applicable for the embedded market where the clock frequency is generally relatively low, thus a series access is possible. It is for these reason we the use the hardware simplicity model. In order to save additional power, some minor modifications must be made to the Execution stage to stop the statically predicted instruction from expending power writing back their results to the predictor (since their results will never be used!).

It can be seen that after a given program has had its hint bits set, all of the branches assigned static predictions (of taken or not taken) have now essentially formed superblocks, with branch resolution acting as a possible exit point from the newly formed super block. When a hint bit prediction proves to be incorrect, it simply acts as a new source of a branch misprediction; it is left for the existing dynamic predictor logic to resolve.

5 Conclusion and Future Work

Branch predictors in modern processors are vital for performance. Their accuracy is also a great source of powersaving, through the reduction of energy spent on misspeculation [8]. However, branch predictors themselves are often comparable to the size of a small cache and dissipate a non trivial amount of power. The work outlined in this paper will help reduce the amount of power dissipated by the predictor hardware itself, whilst not significantly affecting the prediction accuracy. We have begun implementing these modifications in the Wattch [1] power analysis framework (based on the SimpleScalar processor simulator). To test the effectiveness of the modifications and algorithm, we can have chosen to use the EEMBC benchmark suite, which provides a range of task characterisations for embedded processors.

Future investigation includes the possibility of dynamically modifying the hinted predictions contained within instructions to reflect newly dynamically discovered biased branches.

References

1. David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. 27th annual international symposium on Computer architecture, 2000.
2. Colin Egan. *Dynamic Branch Prediction In High Performance Super Scalar Processors*. PhD thesis, University of Hertfordshire, August 2000.
3. Colin Egan, Michael Hicks, Bruce Christianson, and Patrick Quick. Enhancing the I-Cache to Reduce the Power Consumption of Dynamic Branch Predictors. IEEE Digital System Design, jul 2005.
4. Michael Hicks, Colin Egan, Bruce Christianson, and Patrick Quick. HTracer: A Dynamic Instruction Stream Research Tool. IEEE Digital System Design, jul 2005.
5. Erik Jacobsen, Erik Rotenberg, and J.E. Smith. Assigning Confidence to Conditional Branch Predictions. IEEE 29th International Symposium on Microarchitecture, 1996.
6. J. Karlin, D. Stefanovic, and S. Forrest. The Triton Branch Predictor, oct 2004.
7. Alain J. Martin, Mika Nystrom, and Paul L. Penzes. ET^2: A Metric for Time and Energy Efficiency of Computation. 2003.
8. D. Parikh, K. Skadron, Y. Zhang, and M. Stan. Power Aware Branch Prediction: Characterization and Design. *IEEE Transactions On Computers*, 53(2), feb 2004.
9. Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power Issues Related to Branch Prediction. IEEE HPCA, 2002.
10. David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. Morgan Kaufmann, second edition, 1998.