Power Savings in Embedded Processors through Decode Filter Cache [1]

Weiyu Tang Rajesh Gupta Alexandru Nicolau

ICS
Technical Report

Technical Report #-01-63
Sep. 2001

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
{wtang, rgupta, nicolau}@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine

[1]This work was supported in part by DARPA ITO under DIS and PACC program. A version of this paper will be published in Design Automation & Test in Europe, 2002.

Abstract

In embedded processors, instruction fetch and decode can consume more than 40% of processor power. An instruction filter cache can be placed between the CPU core and the instruction cache to service the instruction stream. Power savings in instruction fetch result from accesses to a small cache. In this paper, we introduce decode filter cache to provide decoded instruction stream. On a hit in the decode filter cache, fetching from the instruction cache and the subsequent decoding is eliminated, which results in power savings in both instruction fetch and instruction decode.

We propose to classify instructions into cacheable or uncacheable depending on the decoded width. Then sectored cache design is used in the decode filter cache so that cacheable and uncacheable instructions can coexist in a decode filter cache sector. Finally, a prediction mechanism is presented to reduce the decode filter cache miss penalty. Experimental results show average 34% processor power reduction and less than 1% performance degradation.

Contents

List of Figures

List of Tables

1. Introduction

In embedded processors, often more than 50% area is dedicated to on-chip caches to ensure performance and reduce the number of power expensive memory accesses. Instruction fetch and decode are main consumers of processor power. The power dissipation by different components of StrongARM [5] is shown in Table 1. Instruction fetch and decode together consume 45% processor power. Therefore, they are good targets for power optimization.

instruction cache	27%
instruction decode	18%
data cache	16%
clock	10%
execution	8%
other	21%

Table 1. Power dissipation in StrongARM.

It is well known that small auxiliary structures between the instruction cache (I-cache) and the CPU core can reduce instruction fetch power. Power savings result from accesses to the small and power efficient structures. For example, a line buffer [4] stores the most recently accessed cache line. It utilizes spatial locality in the instruction stream. An **I**nstruction **F**ilter **C**ache (IFC) [6] stores multiple instruction cache lines. It utilizes both spatial and temporal locality in the instruction stream.

In this paper, we introduce a **D**ecode **F**ilter **C**ache (DFC) to provide decoded instructions to the CPU core. A hit in the DFC eliminates one fetch from the I-cache and the subsequent decode, which results in power savings. There is one key difference between the DFC and the IFC. On an IFC miss, the missing line can be filled into the IFC directly. Subsequent accesses to that line need only to access the IFC. In contrast, on a DFC miss the missing line cannot be filled into the DFC because the decoded instructions in this line are not available. As a consequence, the DFC cannot utilize the spatial locality in the missing line. To enable instruction fetch power savings on DFC misses, we use a line buffer in parallel with the DFC to utilize spatial locality of instructions missing from the DFC.

There are several problems with the use of DFC, such as variable width of decoded instructions and performance degradation in case of DFC misses. To make efficient use of cache space, we propose to classify instructions into cacheable or uncacheable. Only instructions with small decode width are cacheable. Then sectored cache design is used in the DFC so that cacheable and uncacheable instructions can coexist in a cache sector. Lastly, we propose an accurate prediction mechanism to dynamic select line buffer, DFC, or I-cache for the next fetch.

The rest of this paper is organized as follows. In Section 2, we briefly describe related work on power savings in instruction fetch and decode. We present in Section 3 the design of DFC. The experimental results are given in Section 4. The paper is concluded in Section 5.

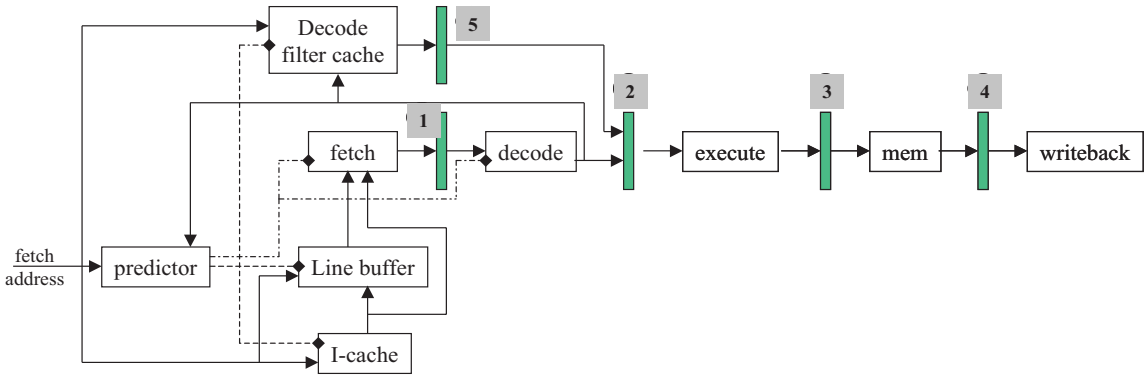

Figure 1. Pipeline architecture

2. Background and Related Work

Several approaches have been proposed to reduce instruction fetch power by using small structures, such as line buffer and IFC, in front of the I-cache. One drawback of IFC is high performance degradation because misses in the IFC will generate pipeline bubbles. Pipeline fetching bubbles can be eliminated based on dynamic prediction. [2] utilizes branch predictor and the IFC is accessed when frequently accessed basic blocks are detected because the IFC hit rate is high for frequently accessed basic blocks. [12] predicts based on the distance between consecutive fetch addresses. The assumption is that the IFC is useful for small loops and the distance between consecutive addresses in a small loop is small.

Instruction decode power reduction through the caching of decoded instructions has also been investigated in previous research. [1] has presented a loop-cache for decoded instructions. It targets DSP processors, which have fixed decode width and tight loops. This approach has difficulties dealing with branches inside loop body, which is common in general purpose embedded processors.

Micro-operation cache [10] also reduces decode power by caching decoded instructions. It adds an extra stage to the pipeline, which increases branch misprediction penalty. The micro-operation cache fill and retrieval is basic block based. This requires a branch predictor that is not necessarily available in most embedded processors such as StrongARM. In addition, the I-cache and the micro-operation cache are probed in parallel. Hence the average per access power is higher than that in our prediction based approach.

[11] goes one step further by saving decoded instructions in scheduled order in a trace cache. It targets high-performance processors where instruction issue is both complex and power consuming. This is overkill for embedded processors where instruction issue is simple. Moreover, the trace cache size is equal or larger than the I-cache. Its high hardware cost is not suitable for embedded processors.

3. Design of Decode Filter Cache

In this section, we address the following problems related to the decode filter cache: (a) how to efficiently save and retrieve decoded instructions with variable widths; (b) how to select line buffer, DFC or I-cache for the next fetch.

3.1. Processor Pipeline

Figure 1 shows the processor pipeline we model in this research. The pipeline is typical of embedded processors such as StrongARM. There are five stages in the pipeline–fetch, decode, execute, mem and writeback. There is no external branch predictor. All branches are predicted "untaken". There is two-cycle delay for "taken" branches.

Instructions can be delivered to the pipeline from one of three sources: line buffer, I-cache and DFC. There are three ways to determine where to fetch instructions:

- serial–sources are accessed one by one in fixed order;
- parallel–all the sources are accessed in parallel;
- predictive–the access order can be serial with flexible order or parallel based on prediction.

Serial access results in minimal power because the most power efficient source is always accessed first. But it also results in the highest performance degradation because every miss in the first accessed source will generate a bubble in the pipeline. On the other hand, parallel access has no performance degradation. But I-cache is always accessed and there is no power savings in instruction fetch. Predictive access, if accurate, can have both the power efficiency of the serial access and the low performance degradation of the parallel access. Therefore, it is adopted in our approach.

As shown in Figure 1, a predictor decides which source to access first based on current fetch address. Another functionality of the predictor is pipeline gating [8]. Suppose a DFC hit is predicted for the next fetch at cycle N. The fetch stage is disabled at cycle $N+1$ and the decoded instruction is sent from the DFC to latch 5. Then at cycle $N+2$, the decode stage is disabled and the decoded instruction is sent from latch 5 to latch 2.

If an instruction is fetched from the I-cache, the hit cache line is also sent to the line buffer. The line buffer can provide instructions for subsequent fetches to the same line.

3.2. Instruction Classification

Decoded instructions may have different widths. If all instructions are allowed to cache in the DFC, then cache line size must be determined by the longest decode width. This may result in cache space underutilization because many embedded processors are designed in RISC style and most instructions have short decode widths.

In order to efficiently utilize the cache space, we classify instructions into cacheable and uncacheable. Only instructions with small decode widths can be cached in the DFC. The classification is done through profiling. First, the execution frequencies of all instructions are obtained from a set of benchmarks. Then execution frequencies of instructions with the same decode width

are summed up. Next, execution frequency table shown in Table 2 is built with increasing order of decode width. Finally, the cacheable ratio, the percentage of dynamic instructions that are cacheable, is selected. This ratio is compared with column "acc_exec_freq" to determine which widths are cacheable.

decode_width	exec_freq.	acc_exec_freq
w_1	f_1	f_1
w_2	f_2	$f_1 + f_2$
...	...	...
w_i	f_i	$\sum_{k=1}^{i} f_k$
...	...	...
w_n	f_n	$\sum_{k=1}^{n} f_k$

Table 2. Decode width frequency table.

3.3. Sectored Cache Organization

In conventional cache designs, one line can have several instructions and the instructions share a tag. For a line of instructions in the next level memory hierarchy, they are either all in the cache, or none of them are in the cache. In contrast, for a line of decoded instructions, some of them may be in the DFC and the rest may be not in the DFC because of cacheable classification. In order to share a tag among these instructions, we use sectored cache design [9] for the DFC.

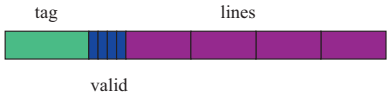

Figure 2. Sector format

A sectored cache consists of several sectors and each sector is made up of several lines. The sector format is shown in Figure 2. All the lines in a sector share one tag and each line has its own valid bit. One disadvantage of sectored cache design is possible cache underutilization because lines corresponding to uncacheable instructions are not used for power savings. A high cacheable ratio can improve cache utilization.

3.4. Prediction Mechanism

Figure 3 shows major components of the predictor for next fetch source prediction. To predict when to access the DFC, we use a **n**ext **f**etch **p**rediction **t**able (NFPT), which is an extension of the approach proposed in [12] with support for sectored cache. The number of entries in the NFPT is equal to the number of sectors in the DFC. Each entry has two fields– *partial_tag* and *sector_valid*. *Partial_tag* is updated using the lowest 4 bits of the tag part of *decode_addr*. The bit in the *sector_valid* that is mapped by *decode_addr* is set to *cacheable*.

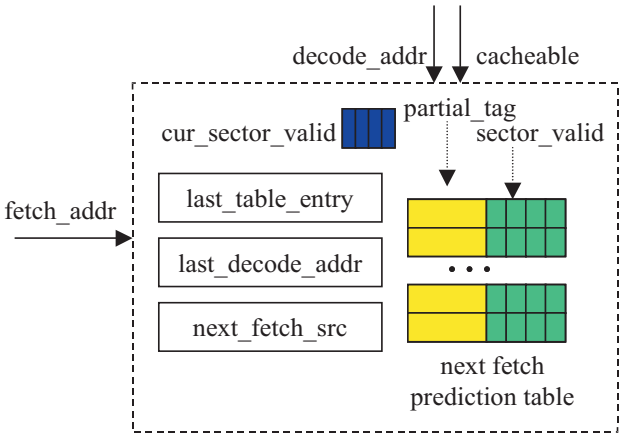

Figure 3. Predictor

The entry to update in the NFPT is pointed by $last\_table\_entry$. If $last\_decode\_addr$ and $decode\_addr$ map to different lines in the DFC, $last\_table\_entry$ is updated using $last\_decode\_addr$. Then $last\_decode\_addr$ is set to $decode\_addr$. Essentially, prediction fields $partial\_tag$ and $sector\_valid$ for current line starting at address cur_line_addr is filled into the entry indexed by the starting address $prev\_line\_addr$ of the previous line. From previous research in branch prediction [14], we know that most branches favor one direction and the same control path may be taken again. Next time when $prev\_line\_addr$ is accessed, the most likely line to be accessed next is cur_line_addr. Therefore, fetch for a line and prediction of the next fetch source for the next line can be done in parallel using the same address $prev\_line\_addr$.

For current fetch address $fetch\_addr$, the most likely next fetch address is $fetch\_addr + 4$. If $fetch\_addr$ and $fetch\_addr + 4$ map to the same cache line, previous fetch source may be reused using the following rules:

1. If $next\_fetch\_src$ is line buffer, then the next fetch will access the line buffer.

2. If $next\_fetch\_src$ is DFC and the corresponding valid bit for $fetch\_addr$+4 in cur_sector_valid is 1, then the next fetch will access the DFC. Otherwise, the next fetch will access the I-cache.

If $fetch\_addr$ and $fetch\_addr + 4$ map to different cache lines, $partial\_tag$ in the NFPT entry indexed by $fetch\_addr$ is compared with the lowest 4 bits of the tag part of $fetch\_addr$. There are two scenarios:

- Equal–Field $sector\_valid$ of the corresponding entry is sent to cur_sector_valid. The $next\_fetch\_src$ is updated as DFC. If the valid bit corresponding to $fetch\_addr + 4$ is 1, then the next fetch source is DFC. Otherwise the next fetch source is I-cache.

- Not equal–The predicted next fetch source is I-cache. However, $next\_fetch\_src$ is updated as line buffer as the line in the I-cache will be forwarded to line buffer.

Mispredictions occur in the following two scenarios:

- Conflict access–Fields *partial_tag* and *sector_valid* in the NFPT have been replaced by conflicting sectors.
- Taken branch–If a taken branch is not at the end of a sector, the prediction for the target address is not available. Otherwise the first valid bit in the *sector_valid* is used in the prediction. However, the target address may be not at the start of a sector and the valid bit for it may be different than the first bit.

4. Experimental Results

4.1. Experimental Setup

Parameter	Value
Instr. size	4B
Line buffer	16B
DFC	direct-mapped, 16 sectors, 4 decoded instr. per sector, 8B per decoded instr.
L1 I-cache	16KB, 4-way, 32B line, 1-cycle latency
L1 D-cache	8KB, 4-way, 32B line, 1-cycle lat.
Memory	30-cycle lat.
IFC	direct-mapped, 32 lines, line size 16B

Table 3. Memory hierarchy parameters.

We use the SimpleScalar toolset [3] to model a single in-order issue processor similar to StrongArm [5]. The memory hierarchy parameters are shown in Table 3. Note that the DFC and the IFC have approximately same hardware cost. We have simulated a set of benchmarks from the MediaBench suite [7]. The description of the benchmarks is shown in Table 4. All the power parameters are obtained using Cacti [13], a tool that can estimate cache power dissipation based on cache parameters such as size, line size, associativity, etc.

The actual width of decoded instructions is highly machine dependent and is not modeled in SimpleScalar. Instead of assigning arbitrary decode width to each instruction and determining the optimal cacheable ratio for a particular processor, we vary the cacheable ratio. Then whether an instruction is cacheable is determined at run-time to satisfy the constraints of the cacheable ratio. In this way, we can evaluate the impact of cacheable ratio on performance and power savings. We have investigated the DFC and IFC configurations shown in Table 5.

Name	Description
721_dec	Voice decompression
721_enc	Voice compress
cjg	Image compression
djg	Image decompression
gst	Ghostscript interpreter
mpg_dec	MPEG decoding
mpg_enc	MPEG encoding
rasta	Speech recognition
adpcm_c	Speech compression
adpcm_d	Speech decompression
epic	Data compression
unepic	Data decompression
pwdec	Public key decryption
pwenc	Public key encryption

Table 4. Benchmark description.

4.2. Results

Figure 4 shows percentage reduction in I-cache fetches. IF has the highest reduction rate. The reduction rate in DF_NO is lower because some instructions are uncacheable, which forces instruction fetch from the I-cache. The reduction rate decreases further in DF_0.9. Even if an instruction is cached in the DFC, the I-cache may still be accessed due to mispredictions. The average reduction for IF, DF_NO and DF_0.9 is 91.4%, 83% and 81.9% respectively. Most fetches to the I-cache are avoided.

Figure 5 shows the percentage reduction in instruction decodes. The reduction rate in DF_0.9 is lower than that in DF_NO due to mispredictions. But the reduction rate by both is very close

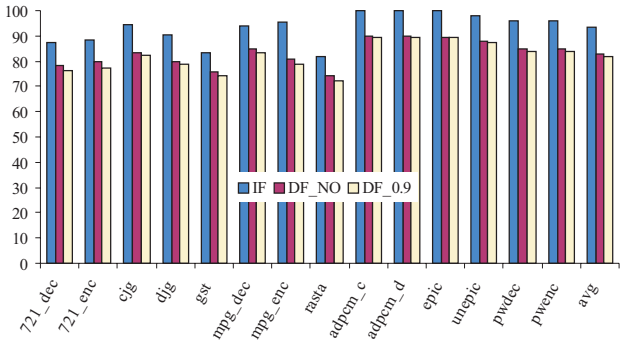

Figure 4. % reduction in I-cache fetches.

	line buffer	DFC	cacheable ratio	IFC	predictor
DF_0.9	√	√	0.9		√
DF_0.8	√	√	0.8		√
DF_0.7	√	√	0.7		√
DF_0.6	√	√	0.6		√
DF_NO	√	√	0.9		
IF				√	

Table 5. Filter cache configurations.

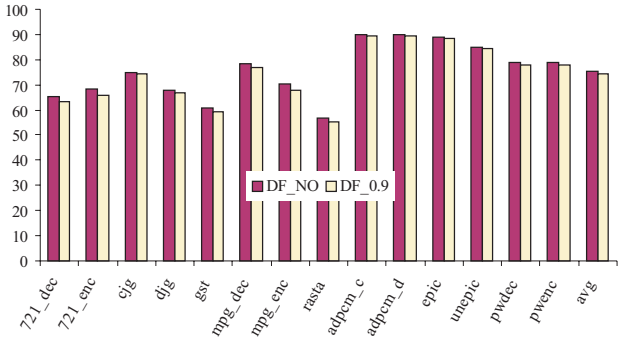

Figure 5. % reduction in instruction decodes.

because the misprediction rate is low. The reduction rate ranges from 56% in *rasta* to 89% in *adpcm_c* and the average rate is 75%. A majority of instruction decodes are eliminated.

Figure 6 shows prediction hit rate in DF_0.9. The minimal rate is 95.3% in *721_dec* and the average rate is 97.7%. This proves that the prediction is highly accurate.

Figure 7 shows normalized delay. The delay increases with the number of DFC/IFC misses. Due to accurate prediction, the number of misses in DF_0.9 is the smallest. Hence the average delay in DF_0.9 is 1.003 and is the lowest. The number of misses in DF_NO is larger than that in IF because of uncacheable instructions. Therefore, the delay in DF_NO is the highest.

Figure 8 shows percentage reduction in processor power. The reduction in DF_0.6 is almost equal to that in IF. The reduction increases with cacheable rate as more number of I-cache fetches and instruction decodes can be eliminated. The average reduction in IF, DF_0.6, DF_0.7, DF_0.8 and DF_0.9 is 23.4%, 23.9%, 27.5%, 31.2% and 34.4% respectively. DF_0.9 is the most power efficient and results in roughly 50% more power savings than IF does.

5. Conclusion

In this paper, we have proposed a decode filter cache, which results in 50% more power savings than an instruction filter cache and the average reduction in processor power is 34%. At the

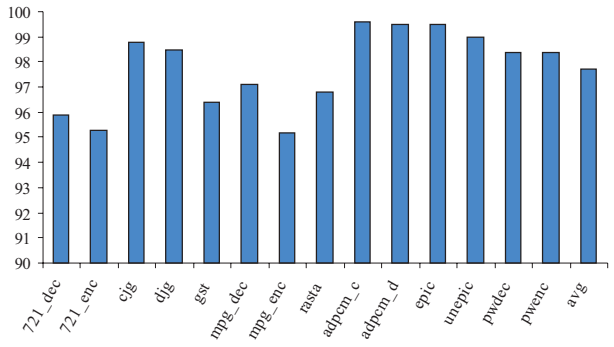

Figure 6. Prediction hit rate in DF_0.9.

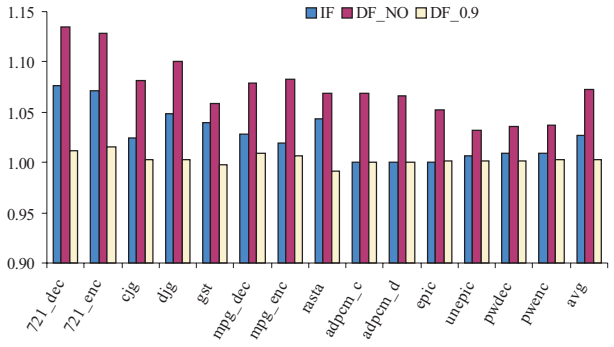

Figure 7. Normalized delay.

same time, the performance degradation is less than 1% due to an accurate prediction mechanism. Comparing to other decoded instruction caching techniques, our approach is simple and the hardware cost is low, which makes it attractive for embedded processors. We are currently extending the decode filter cache design to support multiple-issue processors.

References

[1] T. Anderson and S. Agarwala. Effective hardware-based two-way loop cache for high performance low power processors. In *IEEE Int'l Conf. on Computer Design*, pages 403–407, 2000.

[2] N. Bellas, I. Hajj, and C. Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *Int'l Symp. on Low Power Electronics and Design*, pages 64–69, 1999.

[3] D. Burger and T.Austin. The simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, 1997.

[4] K. Ghose and M. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Int'l Symp. on Low Power Electronics and Design*, pages 70–75, 1999.

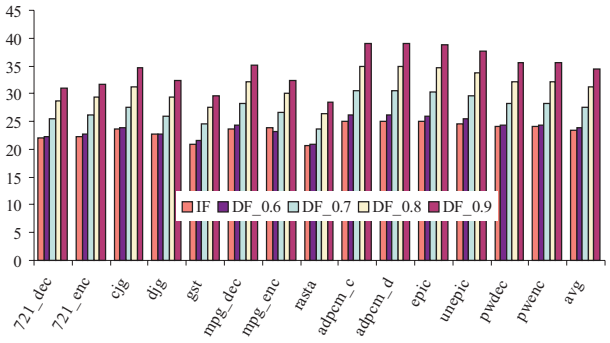

Figure 8. % reduction in processor power.

[5] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 32(11):1703–14, 1996.

[6] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Int'l Symp. Microarchitecture*, pages 184–193, 1997.

[7] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Int'l Symp. Microarchitecture*, pages 330–335, 1997.

[8] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Int'l Symp. Computer Architecture*, pages 132–141, 1998.

[9] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *Int'l Symp. Computer Architecture*, pages 384–393, 1994.

[10] B. Solomon, A. Mendelson, D. Orenstein, Y. Almog, and R. Ronen. Micro-operation cache: a power aware frontend for the variable instruction length isa. In *Int'l Symp. on Low Power Electronics and Design*, pages 4–9, 2001.

[11] E. Talpes and D. Marculescu. Power reduction through work reuse. In *Int'l Symp. on Low Power Electronics and Design*, pages 340–345, 2001.

[12] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Int'l Conf. on Computer Design*, 2001.

[13] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Western Research Laboratory, 1994.

[14] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Int'l Symp. Computer Architecture*, pages 67–76, 1993.