Custom Instruction filter Cache Synthesis for Low-Power Embedded Systems

Kugan Vivekanandarajah and Thambipillai Srikanthan
Centre for High Performance Embedded Systems
School of Computer Engineering
Nanyang Technological University, Singapore.
{kugan@pmail.ntu.edu.sg, astsrikan@ntu.edu.sg}

Abstract

Filter cache has been shown to substantially reduce the power consumption in instruction memory hierarchy. Filter cache achieves energy savings due to the locality found in the frequent tiny loops, which are application dependent. In this paper we show that tuning filter cache to the needs of a particular application can save power and energy. Beside, a simple loop profiler directed methodology to deduce the optimal or near-optimal filter cache is proposed, without having to simulating all possible combinations of cache parameters from the specified space. Our experiments with MediaBench benchmark suite shows that the proposed methodology results in up to 49% energy reduction by tuning the filter cache. Moreover, the proposed filter cache tuning is done with the loop characteristics of the application, which in most cases are readily made available.

1. Introduction

Embedded systems are often designed with tight energy, cost and performance budget. With deep sub-micron (DSM) process technology, the number of transistors devoted for cache memory is increasing, thus, resulting in even higher on-chip power consumption. Much of the dynamic power of a typical embedded processor is consumed by instruction fetches, for example 30-50% [9, 10]. As instruction fetching happens on almost every cycle, it involves switching of a large number of high capacitance wires, and may involve access to a power hungry set-associative cache.

Various auxiliary cache hierarchies have been proposed to alleviate the problem of increased dynamic energy dissipation in the instruction cache hierarchy. These auxiliary cache structures exploit the immediate spatial and temporal locality within small loops of applications. Line buffer[26, 27], loop cache [11, 12] and filter cache [13] are examples of such cache memories. They perform in slightly different ways to extract the energy savings possible by introducing a small cache between the execution core and the main cache. Filter cache, the most effective in energy reduction, has been shown to achieve 58% power saving at the cost of 21% performance loss. Adding predictive [15, 16, 14] capability to the filter cache hierarchy reduces the performance penalty associated with the hierarchical access of the filter cache. Energy savings in this auxiliary cache comes from accessing the energy efficient auxiliary cache, which is physically placed close to the execution core.

Due to the high spatial and temporal locality exhibited by loops in the instruction stream, these auxiliary structures are accessed more often. Thus, the optimal filter cache which results in the maximal energy savings depends mainly on the loop characteristics of the application. Therefore, it is not possible to develop a filter cache configuration which produces the best possible result for all the applications.

However, embedded systems typically run one fixed application for the system's lifetime. Furthermore, embedded system designers are increasingly utilizing microprocessor cores rather than off-the-shelf microprocessor chips. The combination of a fixed application and a flexible core provides the opportunity to tune the architecture of the core to that fixed application.

Architecture tuning is the process of customization of an architecture to most efficiently execute a particular application (or set of applications) under given constraints on size, performance, power and energy. However, it is not feasible to simulate all possible combinations of parameter from the specified space. Because of the multi-dimensional design space, the total number of possible designs can be exceedingly large.

Therefore in this paper we have proposed a methodology for custom instruction filter cache synthesis using loop profiling for core based embedded systems. Loop profiling is often done in embedded systems for different purposes such as hardware software partitioning[22] of the system. Thus with the readily available loop statistics, one can design the near-optimal filter cache hierarchy which results in substantial energy savings.

This paper is organized as follows: next section presents the related research. In the subsequent section, we introduce the predictive filter cache and the energy reduction possible with the predictive filter cache. In Section 3 we present the loop profiler for the linked binary. And Section 4 presents the algorithm for filter cache selection based on the loop profiler output. Section 5 presents the results and finally Section 6 concludes the paper.

2. Related Research

Traditionally simulation based exhaustive search methods are used to explore the optimal cache designs, where a design is simulated with all the possible parameters and analyzed to find the optimal cache hierarchy. But the time required for an exhaustive search is often prohibitive. When the design space becomes too large, iterative heuristics[1] are used to reduce the design space needed to be explored, so that the near-optimal cache configuration is reached without actually simulating the entire configuration. The most crucial to this design methodology is the simulation speed. Thus, some techniques focused on reducing the time required by using trace compression and /or simulating multiple configurations[20, 21] in a single pass. In trace compression, a reduced trace is obtained which approximates the behaviors of the original trace. Techniques are formulated to find approximate[2] or lossless trace reduction [3, 4, 5] to improve the simulation speed.

Another class of approach uses analytical design space exploration, where certain characteristics are extracted from the application and used to either find the miss rate for given cache parameters or, to find the optimal cache parameters for given set of requirements. In data caches, source code analyzing method is proposed to determine the optimal data cache size by analyzing array access patterns of the application[6]. This was later extended to decide upon the distribution of the on-chip data memory into scratch-pad memory and data caches[7]. In addition, an analytical algorithm also was proposed to analyze the data memory references and it is guaranteed to satisfy designer provided performance constraints [8]. Reuse distance[24] was also used to estimate the capacity miss rates of a fully associative cache [24].

In contrast our proposed methodology, which is used to find the near optimal filter cache for instruction cache hierarchy uses loop characteristics of the application and its execution, which in most cases are readily available. In earlier research [25], loop profiling information is used to find the application specific custom loop caches. The working of loop caches are different from the normal cache operation and, its working is more straight forward to predict.

The next section provides the background for the filter cache based memory hierarchy before presenting our proposed methodology.

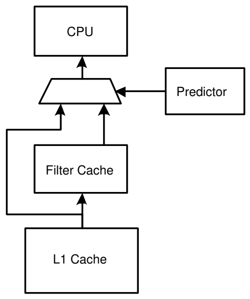


Figure 1. Predictive Filter Cache

3. Background: Filter Cache

Benchmark	128	256	512	1024	2048
ADPCM-DEC	0.60	0.61	0.32	0.31	0.34
ADPCM-ENC	0.58	0.59	0.45	0.31	0.34
EPIC	0.31	0.29	0.29	0.32	0.35
UNEPIC	0.38	0.33	0.31	0.32	0.35
G721.DEC	0.45	0.44	0.40	0.42	0.45
G721.ENC	0.46	0.45	0.40	0.43	0.46
GSM.TOAST	0.51	0.51	0.47	0.40	0.38
GSM.UNTOAST	0.51	0.51	0.42	0.32	0.35
MIPMAP	0.38	0.39	0.39	0.38	0.40
OSDEMO	0.53	0.47	0.37	0.34	0.37
TEXGEN	0.52	0.50	0.43	0.42	0.42
MPEG2.DEC	0.37	0.36	0.36	0.36	0.36
MPEG2.ENC	0.47	0.42	0.33	0.31	0.35
PEGWIT	0.40	0.35	0.31	0.33	0.35
PEGWIT.DEC	0.40	0.33	0.31	0.33	0.35
PEGWIT.ENC	0.42	0.34	0.33	0.34	0.37
JPEG.DEC	0.41	0.41	0.37	0.34	0.36
JPEG.ENC	0.36	0.36	0.33	0.34	0.37

Table 1. Normalized Instruction Access Energy for MediaBench

Filter cache[13], a small direct mapped auxiliary cache, has been shown to be effective in reducing the energy consumption. Though the resulting energy-delay product reduction due to the filter cache is significant, the performance degradation due to the sequential nature of the accesses to L1 cache in cases of a miss in the filter cache might not be acceptable in certain applications. It has been reported that having a predictor based parallel access path between processor core and the filter cache or main cache [13] could reduce the performance penalty. The prediction algorithm predicts whether a particular access is going to be hit or miss in the filter cache. Following the prediction, a direct access is made to the L1 cache or the filter cache. Figure 1 shows the predictive parallel access path present in the predictive filter cache.

Hit/miss predictors predict whether a particular instruction access will be a "hit" or "miss" in the filter cache. Thus it reduces the spurious accesses to the filter cache which might have otherwise resulted in misses. This direct access path to the L1 cache in-turn reduces the performance penalty. Earlier research with filter cache predictors reveal that the *pattern predictor* [14] results in improved prediction accuracy over Next Fetch Prediction Table (NFPT) [15, 16] with minimal overheads in hardware complexity. A detailed study comparing NFPT and the pattern predictor can be found in[14].

Table 1 shows the normalized energy for the applications from MediaBench[19] for various filter cache sizes. The normalization is done with respect to the case where filter cache is not present. Here, SimpleScalar [17] toolset with ARM Instruction Set Architecture (ISA) is used for these experiments. To estimate the access energy per instruction, CACTI version 3.0 [18] is used. The energy consumption of the predictor logic is estimated using Synopsys Power Compiler. The processor model simulated, is a single issue processor typically found in general purpose embedded processors where instructions are fetched from the instruction cache hierarchy every clock cycle. If an instruction cache access results in a miss then it leads to a pipeline bubble. Also, the first level instruction access is in the critical path. L1 cache used in the experiment is 32 KB with 32 bytes line size and 4 way set-associative. The data array is sub banked into 2. This is strictly taking on chip memory accesses into consideration and L1 cache assumed to be on chip as in typical embedded processors.All the applications were compiled with gcc 2.93 and were run to completion. The entire filter cache configurations used in these experiments are with 16-byte line size and direct mapped configuration. The pattern predictor is simulated with a 32 entry PHT.

It is clear from the tables that the predictive filter cache results in substantial energy reduction. The predictive filter cache achieves this energy savings with negligible performance loss (average 2.6% for the benchmarks simulated with 512 Byte filter cache). It is clear from the Table 1 that the optimal energy reduction depends on application, as various benchmarks results in optimal filter cache with different filter cache sizes. Thus, the application profiling for studying the characteristics of the application is crucial to determine the optimal or near-optimal configuration in the filter cache based instruction cache hierarchy. The next section presents the methodology we used to extract the loop characteristics of the applications.

4. Loop Profiling

Loop profiling is often done in embedded systems for different purposes such as hardware software partitioning and optimization in embedded systems. Compilers frequently use profile-directed optimizations to find the hot-paths within the applications to perform optimizations. Thus loop profiles for the embedded applications are readily available in most cases. When the loop statistics needed for the application is not available, the linked binary of the applications can be profiled to extract the applications.

Loop profiler, for the extraction of the loop profile for the executable, can be implemented either by instrumenting the binary for extracting the information or by executing the application on the instruction set simulator. In the second method, though slower, the binary is executed unmodified, leading to more accurate results. Thus, our loop profiler is implemented using the second method. Our implementation, when coupled with SimpleScalar[17] instruction set simulator generates the loop profile of a given ARM program.

Algorithm 1 computes the list of basic blocks for each loop

```
1:  n,p: Node
2:  N : list of all frequent nodes
3:  for $each\ loop\ (detected\ by\ loop\ backedge)$ do
4:    n:=loop_head
5:    while 1 do
6:      add n to the list of node for this loop
7:      for $each\ p\ \epsilon\ of\ Succ(n)$ do
8:        if (p Precedes loop_tail) AND p != loop_tail then
9:          push(p)
10:         Add p to the loop_list (if already not there)
11:       end if
12:     end for
13:     n=pop()
14:     if n=empty then
15:       break
16:     end if
17:   end while
18: end for
```

The loop profiler constructs the Control Flow Graph (CFG) for the whole program, which is equivalent to a call graph plus the CFG for each function, when the application being executed in the simulator. Then, when the program is being executed in the simulator, the edge profile for the application is also constructed. Once we have the edge profile, we strip the nodes with edge counts less than a minimum value from the CFG. This is done so that the number of entries in the CFG is minimized. This is important as uninterested nodes in the CFG can be removed, resulting in improved runtime. Once we get the stripped CFG with the edge profile, we can detect the loops in the control flow by looking at the back edges which is not a "function call" "or function return". Algorithm-1 explains how the nodes which constitutes the loop can be found. It should

Algorithm 2 creates precedence list (used to detect if there is a path between two basic blocks)

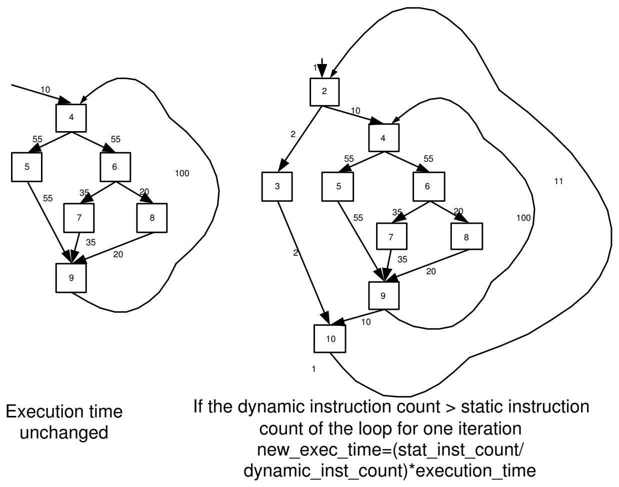
```
1: D,T : in set of Node
2: n,p : Node
3: N : list of all frequent nodes
4: change := True : boolean
5: Precede : Node → set of Node
6: for $each\ \ n\ \ \epsilon\ \ of\ \ N$ do
7:     Precede(n) := {}
8: end for
9: repeat
10:     change := false
11:     for $each\ \ p\ \ \epsilon\ \ of\ \ Pred(n)$ do
12:         T ∪= Precede(p)
13:     end forD:={n}∪ T
14:     if D !=Preceed(n) then
15:         change := TRUE
16:         Precede(n):= D
17:     end if
18: until $!change$
```

be noted that, for cache size studies, we are interested in all the loops (strongly connected graphs) as against natural loops which are detected based on dominator tree[23]. Thus, the Algorithm-2 constructs a list of precedence node list for each, which will be used to check whether there is a path, and Algorithm-1 then traverse the CFG from the loop head to find all the nodes which constitutes the loop. The following sections show the profiling information we extract and the methodology used for determining the cache size.

5. Filter cache size Detection Methodology

The energy savings in the filter cache hierarchy comes from the frequent tiny loops which are cached in the filter cache till the next iteration. While executing larger loops, or while executing the instruction stream which does not have immediate temporal locality, filter cache acts as a line buffer. When the filter cache acts as a line buffer, the size of the filter cache does not matter as a single line itself is sufficient to exploit spatial locality. On the other hand, the per-access energy of the filter cache increases gradually when the filter cache size is increased.

The basic requirement of filter cache for energy savings is to completely or partially cache small but frequent loops while they are executed. Thus, the filter cache size should be large enough to cache these frequent loops completely. If the size of the filter cache is increased, more and more frequent loops can be completely cached in the filter cache, thus, resulting in reduced L1 cache accesses, which in-turn will reduce the overall cache access energy. However, increasing the filter cache size increases the per-access energy and the static energy (due to bigger cache array). Increasing the filter cache over a limit would increase the filter cache access energy and negate the advantage possible with the filter cache. Therefore we designate distinct filter cache sizes which might take advantage by acting as a filter cache. In our case we use filter cache sizes of 128 Byte, 256 Byte, 512 Byte, 1024 Byte and 2048 Byte. Thus for selecting the appropriate filter cache, we need to calculate the relative energy savings possible for the given application with the each filter cache and select the filter cache size with maximum savings. It should be noted that we are interested in calculating the relative energy savings and not the actual savings.



10
4
55
55
5
6
100
55
35
20
7
8
35
20
9
Execution time unchanged
2
10
2
4
3
55
55
5
6
11
55
35
20
100
7
8
2
35
20
9
10
10
1
If the dynamic instruction count > static instruction count of the loop for one iteration
new_exec_time=(stat_inst_count/ dynamic_inst_count)*execution_time

Figure 2. Effective Execution Time Calculation for Inner and Nested Loop

Filter cache exploits the spatial locality by acting as a line buffer. This results in the energy savings due to the spatial locality in instruction access. Assuming that all the instructions are executed in a cache line (16 byte for filter cache), 75% of the instruction accesses will be hit in the filter cache, when only spatial locality is possible. In addition, filter cache also adds a slight energy overhead in-terms of write-accesses to the filter cache for each new cache line access of L1, which is subtracted as shown in the subsequent equation (Equation 1). Since the spatial locality will be possible in all the cache line. Thus, it is being multiplied by 1.0 (100% of the time) to account that it is possible throughout the execution time. Here, EL1 and EFC stand for the access energy for the L1 cache and the filter cache.

$$ES = ((0.75) \cdot EL1 - EFC) \cdot (1.0) \qquad (1)$$

Apart from the spatial locality, the main reason for employing filter cache is to get energy savings within the frequent tiny loop depending on whether the loop can be completely/partially cached in the filter cache till the next iteration. Thus, to compare the energy savings possible due to the temporal locality in the filter cache, we need to know about the loops which are completely or partially cached in the filter cache till the next iteration. We also need to know about the execution profile/frequency of these loops. When we have the loop profile, which comprises the loop size and the percentage of the total execution time being spent on these loops, we need to calculate the relative energy savings for each loop and then normalize it with the percentage execution time for this loop. For this, we first need to know the filter cache size needed to cache all the instruction of a loop, assuming there is no conflict misses between the instructions in the loop. This can be very easily done as we have the list of basic blocks which constitutes this loop.

We also differentiate the inner loops and nested loops for energy savings calculations. That is, in the nested loop, though the outer loop execute less frequently, it might have higher execution time(et) because of the inner loops. For instance, if we look at Figure-2, it executes less frequently and it is not necessary to increase the filter cache size to accommodate the loop. Thus, when we calculate the execution frequency for outer loops, if the dynamic instruction count of the loop is higher than the static instruction count of the loop then we multiply the execution frequent by $\frac{Static\_instruction\_count}{Dynamic\_instruction\_count}$ to consider this.

The filter cache size needed to cache a particular loop is generally larger than the loop size. This is because, the loop might not be placed in the continuous memory location and the basic blocks in the loop have to be aligned to the instruction cache lines as shown in Figure-3. Since, the line size of our filter cache is 16 byte, we align the instructions in the loops to 16 byte and calculate the filter cache size needed to cache all of them. For example, if we have a loop shown as in Figure-3, we will need a slightly larger filter cache to completely cache the loop. And also, we cache only the basic blocks which are execute at least 20% of the loop iterations. That is, we accommodate only the basic blocks which are executed frequently. In our subsequent discussions in this section, whenever we refer to loop size, we will be referring to the filter cache size needed to cache the loop (with the frequent basic blocks) in the direct mapped filter cache.

If the loop size is more than 2·FC (FC here and in the subsequent discussion stands for filter cache size), then there will not be any temporal locality possible with this filter cache. This is assuming there is no conflict misses between instruction lines which constitute this loop. Since, we are looking into tiny loops which are being cached in the small filter cache, this assumption generally holds true. If the loop size is less than that of the FC, a complete loop can be cached in the filter cache, and then full energy savings will be extracted with the filter cache. On the other hand, if the loop size is between 2·FC and FC, then, partial energy savings will be extracted from the filter cache. Here, "et" stands for the fraction of the total execution time spent in this loop, which accounts for the inner and nested loop effect which we have already discussed. Thus:

$$ET = \begin{cases} = 0 & \text{if } loop > 2 \cdot FC \\ = (0.25) \cdot EL1 \cdot et & \text{if } loop \leq FC \\ = (0.25) \cdot (1 - (\frac{loop - FC}{loop})) \cdot EL1 \cdot et & \text{if } loop < 2 \cdot FC \end{cases} \quad (2)$$

The total relative energy savings for a given filter cache over a conventional filter cache is the addition of ET and ES. Algorithm-3 shown below calculates the energy savings with the given filter cache. Once we calculate the relative energy savings with the different filter cache sizes, the one with the maximal energy savings is selected as the suitable filter cache for the given application. The following section produces the results obtained with the proposed analytical model.

Algorithm 3 Calculate the energy savings with the filter cache of size FC

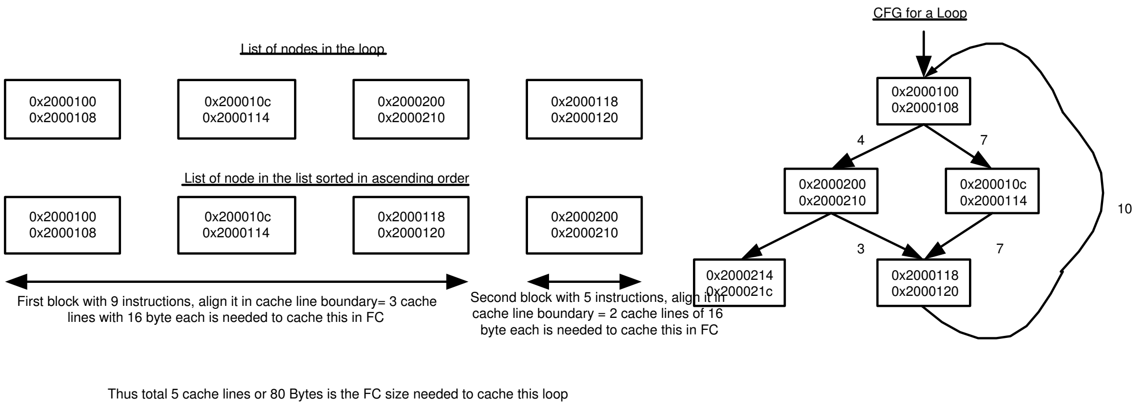
```
1:  $savings$= (0.75)·EL1-EFC
2:  for $loop = CriticalLoop_1$ to $CriticalLoop_N$ do
3:    if $loop > 2 \cdot FC$ then
4:      $savings$+=0
5:    else
6:      if $loop <= FC$ then
7:        $savings$+=(0.25)·EL1·et
8:      else
9:        if $loop < 2 \cdot FC$ then
10:         $savings$+=(0.25) · (1-($\frac{loop-FC}{loop}$)) · EL1 · et
11:       end if
12:     end if
13:   end if
14: end for
```

6. Results for filter cache size detection

Figure 4 compares the relative access energy obtained with the analytical model to that obtained with the simulation, for few benchmarks. The analytical model, which is shown earlier, produces the relative savings per access. Thus, it is subtracted from the per access L1 energy, to obtain the average per access energy with the filter cache. Both the calculated and simulated and the calculated are normalized such that the minimum is one. As can be seen from Figure 4, the results obtained and the analytical model tracks closely, as expected.

Table 2 compares the optimal filter cache size for energy reduction obtained by experimental method with that obtained with analytical model. Figure-5 compares the normalized energy for optimal filter cache configuration, calculated filter cache configuration and a fixed filter cache configuration of 256 byte for all the applications.

As it can be seen from the Table-2, analytical model produces the best filter cache configuration for most of the benchmarks. That is, analytical model finds the optimal configuration except for PEGWIT, PEGWIT.DEC and PEGWIT.ENC. Even for the cases where it does not produce the correct results, the difference in the energy re-



List of nodes in the loop
0x2000100 0x2000108
0x200010c 0x2000114
0x2000200 0x2000210
0x2000118 0x2000120
List of node in the list sorted in ascending order
0x2000100 0x2000108
0x200010c 0x2000114
0x2000118 0x2000120
0x2000200 0x2000210
First block with 9 instructions, align it in cache line boundary= 3 cache lines with 16 byte each is needed to cache this in FC
Second block with 5 instructions, align it in cache line boundary = 2 cache lines of 16 byte each is needed to cache this in FC
Thus total 5 cache lines or 80 Bytes is the FC size needed to cache this loop
CFG for a Loop
0x2000100 0x2000108
0x2000200 0x2000210
0x200010c 0x2000114
0x2000214 0x200021c
0x2000118 0x2000120
4
7
3
7
10

Figure 3. Calculating FC Size Needed For Caching A Loop In Filter Cache With 16 Byte Line Size

Benchmark	Optimal filter cache Size	Calculated filter cache Size
ADPCM-DEC	1024	1024
ADPCM-ENC	1024	1024
EPIC	256	256
UNEPIC	512	512
G721.DEC	512	512
G721.ENC	512	512
GSM.TOAST	2048	2048
GSM.UNTOAST	1024	1024
MIPMAP	128	128
OSDEMO	1024	1024
TEXGEN	1024	1024
MPEG2.DEC	512	512
MPEG2.ENC	1024	1024
PEGWIT	512	256
PEGWIT.DEC	512	256
PEGWIT.ENC	512	256
JPEG.DEC	1024	1024
JPEG.ENC	512	512

Table 2. Optimal Filter Cache Size For Energy Reduction

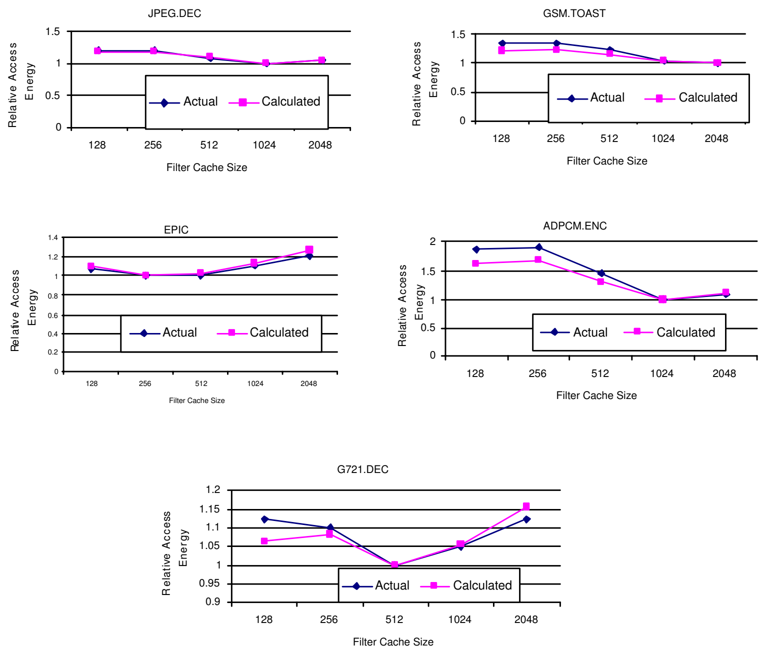


Figure 4. Calculated and Simulated Relative Access Energy Comparison for Filter Cache

duction with the optimal filter cache and the next best filter cache is extremely small (See Table 1). The reason for this is due to the assumptions made in developing the analytical model. We assumed that all the instructions in the filter cache are accessed. However, in practice it is not 100% true, though the instruction accesses have high spatial locality. And also, we assumed that the predictor used in the filter cache is 100% accurate and it is independent of the filter cache size. In practice, though filter cache predictors are highly accurate, they never reach 100%. And also, the prediction accuracy of the filter cache increases with increase in the filter cache size. Overall, tuning the filter cache for the given application results in up to 49% energy reduction as compared to using a fixed filter cache size of 256 Byte. The average gain for the applications simulated is 11% compared to fixed filter cache size of 256 Byte.

7. Conclusion

We have presented a simple yet effective analytical model, based on loop characteristics of the application, which can be used to infer the filter cache configuration which results in near-optimal energy savings for a given application. The proposed methodologies find the near-optimal cache configuration in matter of sec-

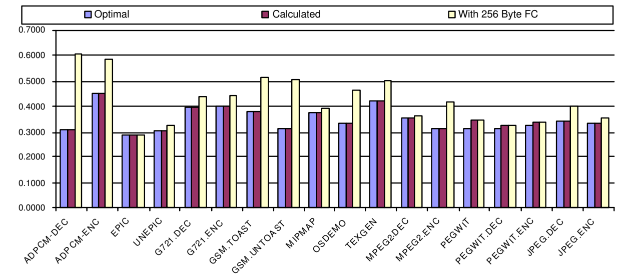


Figure 5. Comparison of Energy Reduction with Optimal filter cache Vs. Calculated filter cache for MediaBench Applications

onds, without actually simulating the filter cache configurations. Thus, the custom filter cache hierarchy synthesis can be seamlessly made part of a core-based embedded system design flow. Our extensive simulations with the MediaBench benchmark suit confirm that the proposed methodology results in 11% energy reduction by tuning the filter cache size as compared to using a fixed filter cache size of 256Byte. The proposed analytical model also serves well to explore and understand the relative energy savings possible with the filter cache hierarchy.

References

[1] Zhang, C., Vahid, F., "Cache configuration exploration on prototyping platforms", 14th IEEE International Workshop on Rapid System Prototyping , June 2003.

[2] S. Laha, J.H. Patel, and R.K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems", IEEE Transactions on Computers, 37(11), 1988.

[3] T. R. Puzak. Analysis of cache replacement algorithms. PhD thesis, University of Massachusetts, Amherst, February 1985.

[4] Wen-Hann Wang and Jean-Loup Baer, "Efficient trace-driven simulation methods for cache performance analysis", In Proc. 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, pages 27-36, May 1990.

[5] Z. Wu and W. Wolf, "Iterative cache simulation of embedded CPUs with trace stripping", In International Workshop on Hardware/Software Codesign, 1999.

[6] P.R. Panda, N.D. Dutt, and A. Nicolau. "Data cache sizing for embedded processor applications", In Design Automation and Test in Europe (DATE), 1998.

[7] P.R. Panda, N.D. Dutt, and A. Nicolau. "Architectural exploration and optimization of local memory in embedded systems", In International Symposium on System Synthesis (ISSS 97), 1997.

[8] A. Ghosh and T. Givargis, "Analytical design space exploration of caches for embedded systems", In Design Automation and Test in Europe (DATE), 2003.

[9] Lee, L. H., Moyer, B., And Arends, "Low-cost embedded program loop cachingrevisited" U. Mich. Technical Report Number CSE-TR-411-99.

[10] Segars, S., "Low power design techniques for microprocessors", In IEEE International Solid- State Circuits Conference, 2001.

[11] Lee L. H., Moyer B. and Arends J.,"Instruction fetch energy reduction using loop caches for embedded applications with small tight loops", In International Symposium On Low Power Electronics and Design, pp. 267-269, 1999.

[12] Moyer B., Lee L. H. and Arends, J., "Data processing system having a cache and method thereof", US Patent number 5, 893, 142, 1999.

[13] Kin J., Gupta M. and Mangione-Smith W.H., "Filtering memory references to increase energy efficiency", Computers, IEEE Transactions on , Volume: 49 Issue: 1, pp. 1 -15, Jan 2000.

[14] K. Vivekanandarajah, T. Srikanthan and S. Bhattacharyya, "Energy-delay efficient filter cache hierarchy using pattern prediction scheme", IEE Proceedings - Computers and Digital Techniques, Vol. 151, Issue 2, March 2004.

[15] Weiyu Tang; Gupta, R.; Nicolau, A.: "Design of a predictive filter cache for energy savings in high performance processor architectures", Computer Design International Conference Proceedings, pp. 68-73, 2001.

[16] W. Tang, R. Gupta, and A. Nicolau, "Power Savings in Embedded Processors through Decode Filter Cache", Design Automation & Test in Europe, pp. 443-448, March 2002.

[17] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set", Technical report, University of Wisconsin, Madison, WI, 1997.

[18] P. Shivakumar and N Jouppi, "An Integrated Cache Timing, Power and Area Model", Tech. Report, Compaq Western Research Lab, Palo Alto, Calif., 2001/2.

[19] Chunho Lee, Miodrag Potkonjak, William H. and Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 330-335, 1997.

[20] M. D. Hill and A. J. Smith. "associativity in CPU caches". IEEE Transactions on Computers, 38(12):1612-1630, December 1989.

[21] R. A. Sugumar and S. G. Abraham. "efficient simulation of multiple cache configurations using binomial trees". Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[22] D.C. Suresh, W.A. Najjar, F. Vahid, J.R. Villarreal, G. Stitt Languages, "Profiling tools for hardware/software partitioning of embedded applications", Compilers and Tools for Embedded Systems (LCTES), 2003, pp. 189-198

[23] Steven P. Muchnick. "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205., 1st edition, 1997.

[24] Y. Zhong, S. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, pages 91-101, New Orleans, LA, September 2003.

[25] S. Cotterell and F. Vahid, "Synthesis of Customized Loop Caches for Core-Based Embedded Systems" IEEE/ACM International Conference on Computer Aided Design pp. 655-662, November 2002.

[26] C. Su, A. Despain, "Cache Design Tradeoffs for Power and Performance Optimization:A Case Study", ISLPED95, pp. 6368, 1995.

[27] K. Ghose, M. B. Kamble, "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-Line Segmentation", ISLPED99, pp. 7075, 1999.