Towards a Performance- and Energy-Efficient Data Filter Cache

Alen Bardizbanyan
Chalmers University of Technology
alenb@chalmers.se

Magnus Själander
Florida State University
sjalande@cs.fsu.edu

David Whalley
Florida State University
whalley@cs.fsu.edu

Per Larsson-Edefors
Chalmers University of Technology
perla@chalmers.se

ABSTRACT

As CPU data requests to the level-one (L1) data cache (DC) can represent as much as 25% of an embedded processor's total power dissipation, techniques that decrease L1 DC accesses can significantly enhance processor energy efficiency. Filter caches are known to efficiently decrease the number of accesses to instruction caches. However, due to the irregular access pattern of data accesses, a conventional data filter cache (DFC) has a high miss rate, which degrades processor performance. We propose to integrate a DFC with a fast address calculation technique to significantly reduce the impact of misses and to improve performance by enabling one-cycle loads. Furthermore, we show that DFC stalls can be eliminated even after unsuccessful fast address calculations by simultaneously accessing the DFC and L1 DC on the following cycle. We quantitatively evaluate different DFC configurations, with and without the fast address calculation technique, using different write allocation policies, and qualitatively describe their impact on energy efficiency. The proposed design provides an efficient DFC that yields both energy and performance improvements.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory structures—*Design styles, Cache memories*

General Terms

Hardware design, execution time improvement, energy efficiency

Keywords

Execution time, energy, memory hierarchy, data cache, speculation

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ODES'13, February 23 - 24 2013, Shenzhen, China
Copyright 2013 ACM 978-1-4503-1905-8/13/02...$15.00

1. INTRODUCTION

Mobile phones and tablets are examples of battery-powered devices that are always on. These devices not only need to be able to deliver high performance when the user actively performs tasks—such as watching a high-definition movie—but they must also support long standby time, during which the operating system performs maintenance tasks like checking for incoming messages.

A trend in mobile devices is to employ a diversity of processor cores with a few high-performance cores in combination with a supporting low-power core. The performance-oriented cores are enabled only when the user actively performs tasks on the device, while the low-power core handles background tasks during long idle periods. NVIDIA employs this concept in their Tegra 3 devices, in which they use two different implementations of a Cortex A9 [24]. ARM use a combination of an in-order Cortex A7 and out-of-order Cortex A15 cores in their big.LITTLE [13] systems.

Energy efficiency is a very important feature for the always-on cores. Recent studies have shown that the power consumed by data requests to the level-one (L1) data cache (DC) constitutes up to 25% of the total power of an embedded processor [11, 15]. Much of the data request power is due to accessing large tag and data arrays in the L1 DC. If we can avoid accessing the L1 DC by accessing a smaller structure that is located closer to the CPU, it is possible to reduce total processor power. However, since these cores often have high performance requirements, such power reductions must not come at the expense of performance.

A considerable amount of research have been conducted to improve the energy efficiency of L1 caches in general. Kin *et al.* proposed the filter cache scheme [21] to reduce the power dissipation of instruction and data caches in an in-order pipeline. The filter cache, which is significantly smaller than an L1 cache, is located between the L1 cache and the CPU pipeline. With a filter cache many memory references are captured in the smaller structure, which saves power as the number of L1 cache accesses is reduced. However, the power and energy reduction of the proposed scheme comes with a significant performance degradation. Due to high miss rates in the filter cache—with each miss causing a one-cycle delay penalty—the performance loss can be substantial. In fact, the performance loss can cancel much of the energy benefits, because of the energy dissipated by the system during the additional execution time.

We propose a new use of the fast address calculation technique [4], to create an access scheme that eliminates the performance penalty previously associated with a data filter cache (DFC). The principal behind our design approach is that the fast address calculation, which has a delay of a single Or-gate, can be performed in the same clock cycle as the DFC access. This enables data accesses to be serviced in the first, address-calculation stage of a conventional two-cycle load/store pipeline. On a DFC miss the L1 DC is accessed as normal in the second stage, thus, eliminating any performance penalties associated with conventional DFCs.

This paper makes the following contributions. (1) We propose a DFC cache design that can completely eliminate the one-cycle miss penalty of conventional data filter caches. (2) We capture a large percentage of the data accesses in a smaller DFC. In addition, we reduce the total execution time, which together can lead to significant energy savings. (3) We evaluate two different allocation policies during write misses and qualitatively describe their impact on energy efficiency.

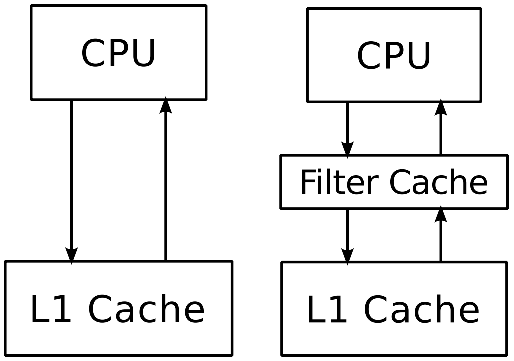
2. BACKGROUND

In this section we review the filter cache and some of its benefits and disadvantages, and then describe the fast address calculation technique and the problems this solves. We will throughout this paper use a conventional five-stage pipeline to illustrate the proposed techniques. The presented techniques are, however, not limited to this classical pipeline, but they can be used for any pipeline where the address calculation occurs in the cycle before the data access. One example of such an architecture is the ARM Cortex A7 that has an in-order, eight-stage pipeline, where the loads and stores are performed across two pipeline stages [18].

2.1 Filter Cache

The DFC has the structure of a conventional cache, however, the number of cache lines that can be stored is significantly smaller. The DFC is inserted into the memory hierarchy between the CPU and the L1 cache, as shown in Fig. 1b. When a request is made, the DFC is accessed and upon a DFC hit, the L1 cache access is avoided. The smaller size of the DFC makes it more power efficient as the memory arrays for storing tags and data are smaller. On a DFC miss, an additional cycle is required to fetch the cache line from the L1 cache into the DFC, while at the same time providing the requested data (or instruction) to the CPU.

Introducing a conventional DFC to an in-order pipeline leads to a performance degradation. Compared to a conventional pipeline with only an L1 DC, a DFC will cause an additional stall cycle for each DFC miss. As the conventional DFC has a high miss rate (as high as 35%), the impact on performance is significant.



(a) Conventional (b) Filter cache

Figure 1: Memory hierarchy with (a) and without (b) filter cache.

Fig. 2 shows the average miss and the write-back[1] rates of different DFC configurations for the MiBench benchmark suite [14] (for benchmark and compilation details, see Sec. 5). The tree-based pseudo least recently used (PLRU) replacement scheme is used for the fully-associative cache and is practical to implement for higher number of ways [1]. While full associativity might not be practically feasible for the larger cache sizes in Fig. 2, it can be suitable for small cache sizes, for which the overhead is modest [7]. Associative filter caches outperform direct-mapped filter caches when employed for data accesses. For example, a 512-byte fully-associative cache has a miss rate of 8.7%, while a 2,048-byte direct-mapped cache has an 8.0% miss rate. Thus, a direct-mapped cache would have to be four times larger in terms of capacity to have a similar miss rate. This result is in contrast to the original study on instruction and data filter caches, which suggested that a direct-mapped filter cache has better performance than an associative filter cache [21], but is consistent with a more recent study [12].

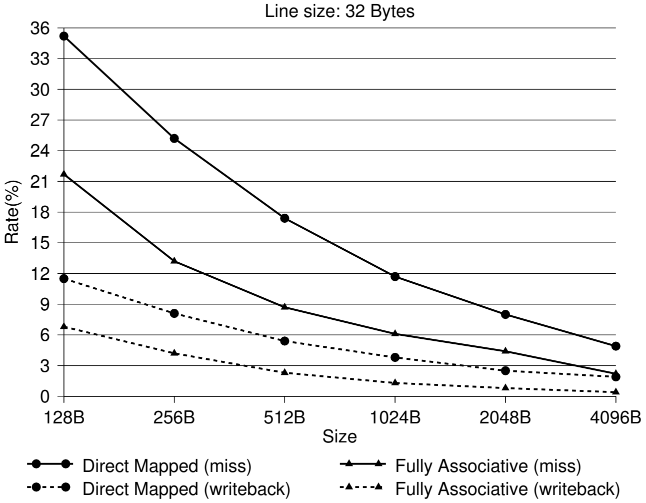


Figure 2: Miss and write-back rates.

[1] The number of write-back operations initiated by the replacement of a dirty line, normalized to the total number of load/store operations.

2.2 Fast Address Calculation

Fast address calculation was proposed as a technique to reduce stall cycles caused by load hazards [4]. Fig. 3 illustrates data memory accesses in a five-stage pipeline. A load operation effectively takes two cycles with the memory address being calculated in the execute (EXE) stage and the memory access being performed in the memory (MEM) stage. In case the instruction following the load depends on the loaded value, as illustrated in Fig. 4, we have what is called a load hazard. The add instruction will be stalled in the execute stage, until the load is completed in the memory stage, at which point the loaded value is forwarded to the add instruction in the execute stage. Load hazards are not infrequent and cause the execution time to increase.

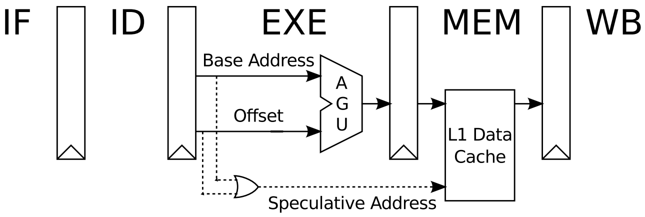
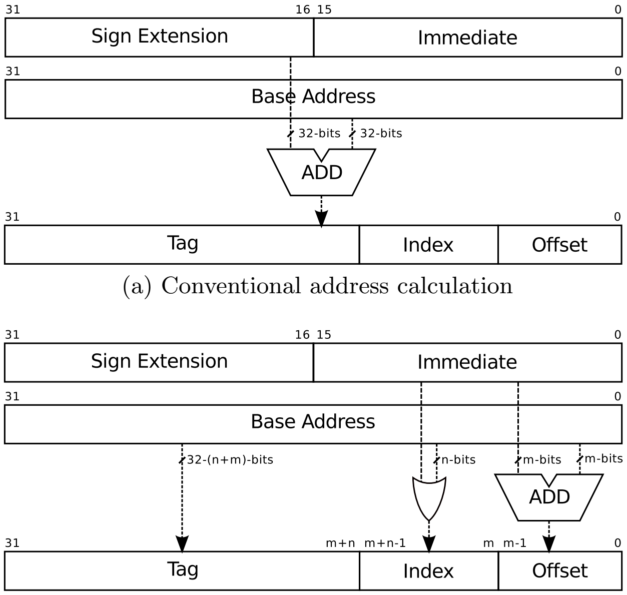


Figure 3: Fast address calculation in a five-stage pipeline.

```
lw $2,10($15)
add $7,$2,$3
```

Figure 4: Instruction sequence that causes a load hazard.

The memory address is commonly calculated by adding an offset (the immediate) to a register value (the base address). An example is shown in Fig. 4, where the offset 10 is added to register $15 for the load operation.



(a) Conventional address calculation

(b) Fast address calculation

Figure 5: Conventional (a) and fast (b) address calculation.

Fig. 5a shows the details of the addition operation for the memory address calculation. It was observed that for most operations, there is no carry out from the line-offset part to the index part of the address, due to the immediate often being a small positive value [4]. In addition, it was observed that when the immediate width is larger than the line offset width, it is still possible to correctly calculate the *index* for a high percentage of accesses by OR'ing the index and the corresponding part of the offset when there is no carry out in any of the bit additions [4]. This also allows a very fast way of calculating the index, since only one level of Or gates is needed. This property of the address calculation operation is used to exploit the SRAM access pattern, so that it is possible to access the L1 DC in the same stage in which the address calculation takes place. The index drives the row decoder that selects the word line, which is on the critical path. The byte offset drives the column multiplexer, which needs to be driven after the word-line selection, hence, the small addition of the byte offset can be tolerated.

3. FILTER CACHE WITH FAST ADDRESS CALCULATION

In this section we describe our design approach, which entails integrating a DFC between the pipeline of the CPU and the L1 DC and using fast address calculation to avoid any performance degradation. The idea is that the DFC performance penalty due to misses can be reduced, or even eliminated, if there is a way to detect a DFC miss in the execute stage. In case a miss is detected, the conventional L1 DC can then be accessed on the following cycle, hence eliminating the stall cycle. The fast address calculation technique makes it possible to speculatively calculate the index and tag with only a single Or-gate delay. The index and the tag can therefore be used to access a DFC directly in the execute stage of the pipeline. At the end of the execute stage, the index and the tag are verified by detecting that no carry into the index portion of the address was created during the conventional address calculation performed by the address generator unit (AGU) [4]. Fig. 6 shows the DFC in the execute stage. It should be noted that for a fully associative cache there is no line index, but the bits after the line offset belong to the tag.

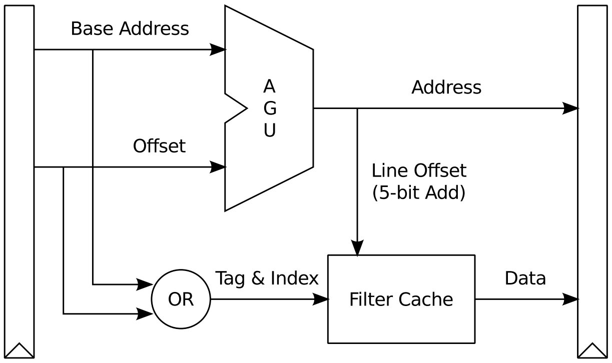


Figure 6: A filter cache with fast address calculation in the execute stage and a cache line size of 32 bytes.

We analyzed the fast address calculation technique on the MiBench benchmark suite (see Sec. 5). Fig. 7 shows the analysis results, assuming a 32-byte line size. On average, for 69% of the store operations and for 74.9% of the load operations, the line index or tag does not change after the address calculation, due to very small immediates. In addition, on average, for 3.1% of the store operations and for 2.4% of the load operations the remaining address after the line offset can be calculated with a single level of Or-gates.

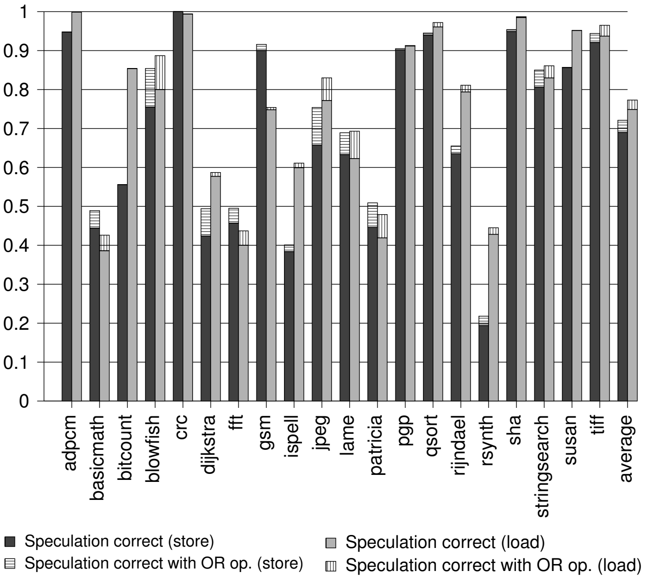


Figure 7: The proportion of correct speculations out of all speculative address calculations.

When the speculation is not successful, that is, when a carry into the index occurred or a carry propagation happened after the line offset, it is not possible to determine if the data resides in the DFC during the execute stage. If the DFC would be accessed in the following cycle to determine if the data resides in the DFC, then there would be a one-cycle penalty if there is a DFC miss. In order to avoid a DFC miss cycle due to a speculation failure, the L1 DC must be accessed on the next cycle after the speculation failure. In addition, if the DFC uses a write-back policy, then both the L1 DC and the DFC must be accessed in parallel because up-to-date data might reside in the DFC only. While this speculation scheme can entirely eliminate the conventional DFC miss penalty, it may not be energy efficient since we will access the L1 DC when there is a possibility that the memory access in fact can be a hit in the DFC. It should also be noted that on a speculation failure or a DFC miss, the one-cycle load cannot be exploited. The impact of all these aspects are evaluated in Sec. 6.

The DFC is virtually tagged to not require data translation lookaside buffer (DTLB) lookups. This reduces data access power and removes the DTLB lookup from the critical path when accessing the DFC. The DFC therefore needs to be flushed on a context switch. The overhead of having to flush the DFC is negligible, due to the infrequency of context switches and the small size of the DFC. On kernel interrupts, the DFC is disabled and all data accesses are serviced directly from the L1 DC. If the interrupt routine accesses data that might be used by the interrupted process, the DFC has to be flushed before accessing this data.

The virtually addressed DFC complicates cache coherency. To efficiently support cache coherency, the L1 DC is inclusive of the DFC contents and the DFC stores the L1 DC set index and way for each of the DFC lines. When an L1 DC line is evicted, the set index and way are checked against those stored in the DFC and if there is a match the corresponding line is also evicted from the DFC. The set index and way are also used when a dirty cache line is evicted from the DFC, to directly write the line back to the L1 DC without requiring a tag check. Storing the L1 DC set index and way for each DFC line is feasible as there are few DFC lines.

4. IMPACT OF WRITE ALLOCATION POLICIES

The choice of the write allocation policy for a cache can have an impact on the expended energy. In fact, the policy choice is even more important in DFCs, because the miss rate is much higher compared to an L1 DC.

Store operations constitute on average 30% of the total memory (load/store) operations across the 20 MiBench benchmarks (Sec. 5). The load operations will constitute a higher percentage of the data cache energy because the rate is substantially higher. As a result, it is desirable to capture most of the load accesses in the DFC. Hence it is preferred to always allocate the line on a miss caused by the load operation. Different allocation policies can be used for the store operations, to handle store misses. These policies are called *write-allocate*, in which a store miss causes a new line to be allocated on the currently accessed cache level, and *no-write-allocate*, in which a store miss does not cause a new line to be allocated, but the write operation is performed at the next level in the memory hierarchy [20].

It is a challenge to reduce store energy when using a DFC; especially if the data cache uses a two-cycle write operation in which the tag match is performed during the first cycle and the data is written during the second cycle. The write power will be less compared to the read power because only one data bank is activated. A DFC can save write power in two ways. First it can gather many write operations into one line, and thus reduce the write operations to the next memory level. In addition, if the DFC is inclusive to the next level, there is no need for a tag check operation during write back, because the way information and line number can be saved during the allocation and these can be used to directly write the data to the data arrays during write back.

Fig. 8 presents the miss rates when the two different allocation policies are used. Here the miss rate directly corresponds to the number of cache lines that needs to be fetched from L1 DC to the DFC normalized to the total number of load/store operations. Hence when the no-write-allocate policy is used, the store misses are not really considered as misses because they will not cause any allocation in the DFC. Thus, the no-write-allocate policy reduces the miss rate. The impact is much higher on the direct-mapped cache. The reason is that the chosen write allocation policy reduces the DFC contention, which improves the direct-mapped cache that is known to suffer from high contention. Reduced miss rates mean less energy dissipation, because there will be less lines fetched from the next memory level. But, It should also be noted that with the no-write-allocate

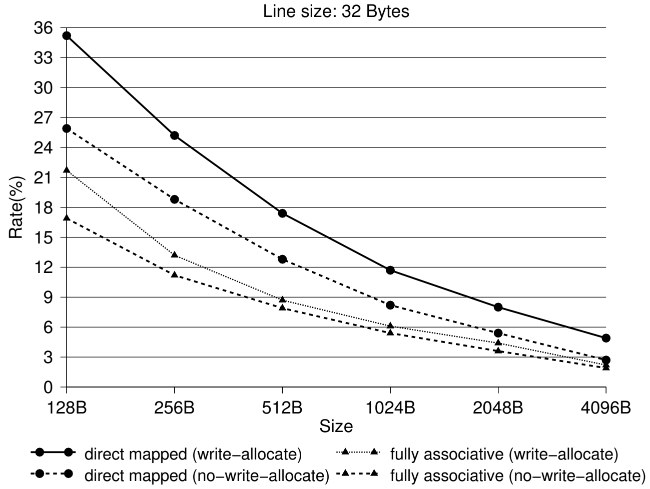


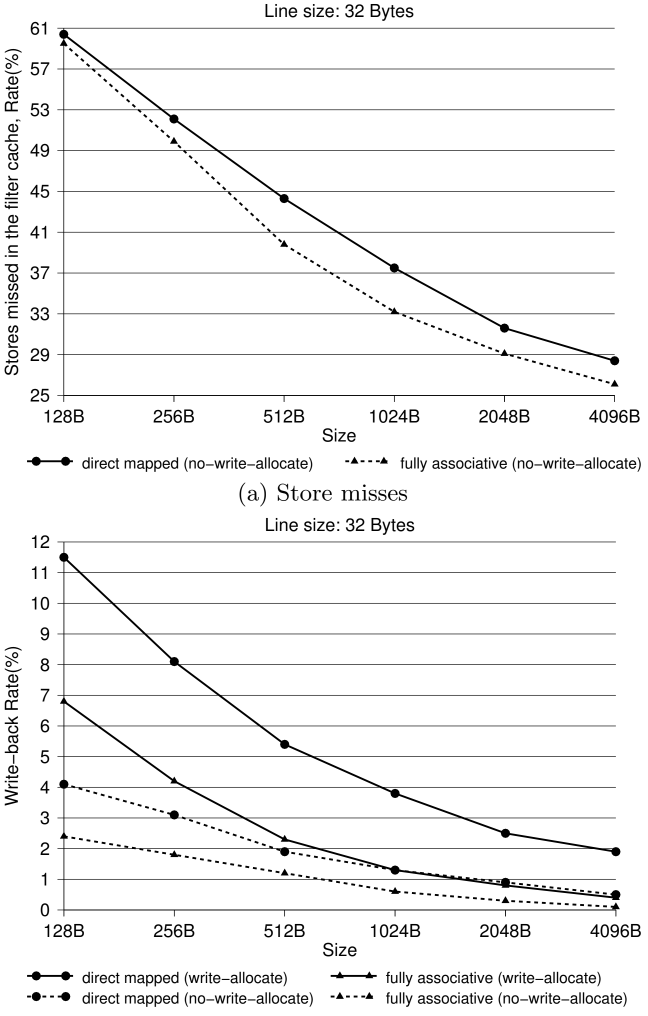
Figure 8: Miss rates depending on the allocation policy.

policy there will be extra direct store operations to the L1 DC for each store miss. As a result the possibility of energy savings depends on the cost of doing a regular store access to the L1 DC and the cost of a line fetch and line writeback oepartions in which the line operations are expected to be more costly.

Fig. 9 shows an interesting trade-off for the store operations when the no-write-allocate policy is used. Fig. 9a shows the number of store operations that miss in the DFC and Fig. 9b shows the write-back rate. For example, in a 512-byte fully-associative cache, 40% of the store operations cause a miss in the DFC and are directly written to the L1 cache. However, the write-back rate decreases from 2.3% for the write-allocate policy to 1.2% for no-write-allocate, which is a 48% decrease. This means more writes are gathered in the DFC before write-back operations, hence more power can be saved in the data bank of the L1 DC for the store operations that have a hit in the DFC. This is even more apparent on direct-mapped DFCs. For example, in a 512-byte direct-mapped cache, 44.3% of the store operations cause a miss the DFC. Here, the write-back rate decreases from 5.4% for the write-allocate policy to 1.9% for no-write-allocate, which is a 65% decrease. Although it takes a detailed power analysis to ascertain which allocation policy is the most energy efficient, this basic analysis shows that the no-write-allocate policy mitigates the disadvantages of using DFCs with lower associativity.

5. EVALUATION FRAMEWORK

In order to evaluate performance, we use 20 different benchmarks (see Table 1) across six different categories in the MiBench benchmark suite [14]. Using the large dataset option, the benchmarks are compiled with the VPO compiler [9]. The SimpleScalar simulator [3] is modified to simulate a time-accurate five-stage in-order pipeline. The configuration of the simulator is given in Table 2. It is assumed that the pipeline stalls on data cache misses, although a store buffer might tolerate some of the miss cycles caused by store misses.



(a) Store misses

(b) Write-back rate

Figure 9: The impact of write allocation policy on (a) store misses and (b) write-back rate in the filter cache.

Previous research showed that a 512-byte fully-associative (FA) cache with a 16-byte line size, implemented in a 45-nm CMOS process tailored to low-standby power (LSTP), dissipates 40% more power compared to a 512-byte direct mapped (DM) cache with the same line size [7]. Since the DFC size is much smaller compared to an L1 DC, also the access power is expected to be much smaller. Hence, the 40% overhead can still make the FA cache an energy-efficient candidate, since it has lower miss rates compared to a DM cache with the same size. However, since the access time will be higher compared to the DM cache, we limit the evaluation of the FA caches to 512 bytes at maximum, using only 16 tags with a 32-byte line size.

6. EXECUTION TIME EVALUATION

In this section we describe the results of our performance evaluations on DFCs. We especially focus on the results of the 512-byte fully-associative (FA) cache and the 2,048-byte direct-mapped (DM) cache, since these have similar miss rates.

Table 1: MiBench benchmarks

Category	Applications
Automotive	Basicmath, Bitcount, Qsort, Susan
Consumer	JPEG, Lame, TIFF
Network	Dijkstra, Patricia
Office	Ispell, Rsynth, Stringsearch
Security	Blowfish, Rijndael, SHA, PGP
Telecomm	ADPCM, CRC32, FFT, GSM

Table 2: Processor Configuration

BPB, BTB	Bimodal, 128 entries
Branch Penalty	2 cycles
Integer & FP ALUs, MUL/DIV	1
Fetch, Decode, Issue Width	1
DFC	128B-512B (FA), 128B-4096B (DM) 32-B line, 1 cycle hit, write-allocate
L1 DC & L1 IC	16 kB, 4-way assoc, 32-B line, 1 cycle hit
L2U	64 kB, 8-way assoc, 32-B line, 8 cycle hit
DTLB & ITLB	32-entry fully assoc, 1 cycle hit
Memory Latency	120 cycles

Fig. 10 shows the execution time for the conventional DFC and for our proposed DFC with fast address calculation. The execution time is normalized to a conventional five-stage, in-order pipeline without a DFC, where the L1 DC is accessed in the memory stage. As expected, the use of the conventional DFC has an adverse impact on execution time. For a 512-byte FA cache this performance penalty is 1.8%, while a four times bigger 2,048-byte DM cache has an execution time penalty of 1.6%.

When fast address calculation is employed, there is a significant gain in execution time due to the avoidance of most of the DFC miss penalties and many load hazards. If we use the scheme where we access the DFC and L1 DC in sequence instead of in parallel on a speculation failure (Sec. 3), then a number of cycles are wasted since we are not able to detect some of the DFC misses early in the pipeline. This overhead is explicitly annotated in Fig. 10. Our further analysis showed that most of the speculation failures happen when the memory operation is in fact a DFC hit. This phenomenon becomes more dominant as the DFC size increases because the miss rate is reduced, which can be observed from Fig. 10. For example, the overhead of accessing them in sequence for a 128-byte DM cache is 1.3%, while it is 0.2% for the 2,048-byte DM cache. In order to reduce this small overhead, the L1 DC needs to be accessed on each speculation failure, which can be very inefficient in terms of energy. Hence it will likely be more energy efficient to only access the DFC again on a speculation failure.

The execution time improvement increases when the DFC size is increased. The reason for this is that more one-cycle loads can be exploited, which is not the case if there is a high miss rate.

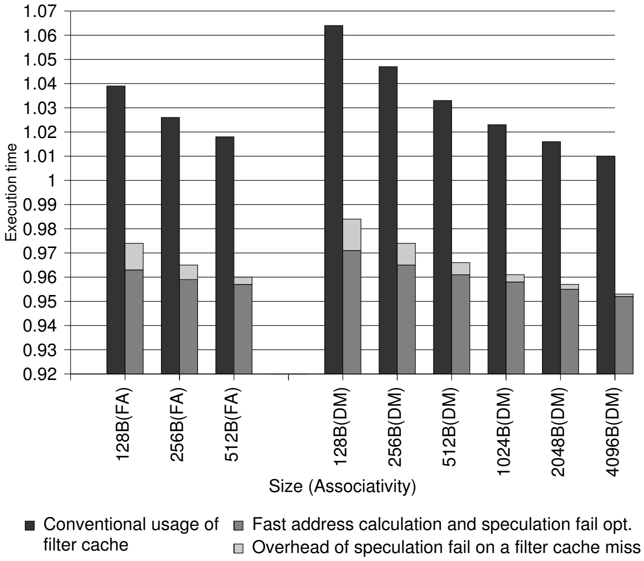


Figure 10: Execution time normalized to a pipeline with conventional L1 DC usage.

When the fast address calculation technique is employed, a 512-byte FA cache provides a 4% execution time improvement when including the speculation failure overhead, while a 2,048-byte DM cache (four times as large) provides a comparable (4.3%) execution time improvement.

We also evaluated the original fast address calculation approach that was used with an L1 DC to eliminate load hazards [4]. This technique can provide a 5.1% execution-time improvement when applied to the reference L1 DC used in this work. The original approach provides a larger execution time benefit due to avoiding more load hazards. The technique reduces the miss penalty by one cycle when the speculation succeeds, as a miss is detected one cycle earlier. However, the original approach is mainly intended for improving performance, as each speculation failure will cause an additional L1 DC access. These extra accesses will increase the L1 DC energy. In contrast, due to the relatively high miss rate of the DFC, the fast address calculation technique provides a greater benefit in combination with a DFC as it eliminates the one-cycle stall penalty on DFC misses and the speculation failures cause an extra access to a smaller structure than the L1 DC.

7. RELATED WORK

Duong *et al.* propose to perform the tag check for the DFC in the same stage as the address calculation happens so that the miss can be detected earlier, as in the case of the fast address calculation [12]. But the tag check takes place immediately after the address calculation and, hence, the critical path includes *both* the address calculation and tag check. Thus, under a strict timing constraint, the address generator and the tag comparison units will require fast and power-hungry circuits, leading to increasing power dissipation for load/store accesses. In contrast, our approach using fast address calculation gives both the address generator unit and the tag comparison unit a full clock cycle to complete, leading to power overhead only during speculation failures. Furthermore, the data array of Duong's DFC is accessed in the next stage after address generation stage [12], which prevents their approach from exploiting early load accesses.

It is possible to distinguish between different data cache access types and implement specific caches for each type [22], however, this complicates resource utilization as the total storage of the cache is split between separate entities. Another approach is to use a predictive technique to read the value from the store queue instead of from the L1 cache [10], leading to a 32% data cache power reduction at a 0.1% performance penalty. One proposed scheme, which claims to eliminate about 20% of the cache accesses, always reads the maximum word size even though the load is only for a byte; the additional data that was read can then be used in a later access [19]. In the vertical cache partitioning scheme [26], a line buffer stores the last read line and is checked before accessing the L1 DC cache. This additional check is on the critical path of the cache and can have a negative impact on both performance and energy. Nicolaescu *et al.* proposed a power-saving scheme for associative data caches [23]. The way information of the last N cache accesses is saved in a table, and each access makes a tag search on this table. If there is a match, the way information is used to activate only the corresponding way. This table needs to be checked before the cache is accessed, hence it can impact the critical path and the memory access latency. In order to eliminate this, Nicolaescu *et al.* propose to use a fast address calculation method to get the way information early in the pipeline stage and eliminate the overhead for the way-table access. Scratchpad memories [5, 25] offer small, energy-efficient memories, however, they have to be explicitly controlled by software. Thus, scratchpads are challenging to manage, as extra code is required to explicitly move data to and from the main memory. Furthermore, they pose challenges when performing context switches.

Instruction accesses have a more regular access pattern and higher locality than data accesses. The regular access pattern of instructions allows special energy saving techniques to be applied to instruction caches. Take, for example, loops where instructions are accessed in consecutive order followed by a backward branch. These regular access patterns have been extensively exploited to improve the hit rate of small structures similar to the filter cache [28, 6, 16, 8]. Also, the regular access pattern has been a key property for way prediction [17] and way selection [2] to make the instruction cache more energy efficient.

8. CONCLUSION AND FUTURE WORK

We presented an approach to designing performance- and energy-efficient DFCs. The impact of miss penalties of a conventional DFC can be reduced by integrating a speculation technique based on fast address calculation to enable one-cycle loads. As a result, for example, a 512-byte fully-associative DFC with a 32-byte line size yields an overall performance gain of 4%. The remaining impact of miss penalties can be eliminated by always accessing the L1 DC on an address speculation failure. In addition, a 512-byte fully-associative DFC can filter 89% of the L1 DC accesses, while the remaining 8.7% are line fetch operations from the L1 DC and 2.3% are write-back operations to the L1 DC.

As far as future work, an energy evaluation using a placed and routed design would reveal the most energy-efficient DFC configuration(s). In addition, optimizing the L1 DC for line fetch operations might improve the energy efficiency when a DFC is employed. DFCs can in particular provide benefits on designs optimized for low standby power (that is, low leakage power) [27], for which the energy will be dominated by the dynamic switching activity. In these designs, one design dimension that can be explored to reduce energy would be the circuits that have substantial switching activities during the stall cycles caused by the DFCs.

9. REFERENCES

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proc. 42nd Annual Southeast Regional Conf.*, pages 267–272, 2004.

[2] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proc. 32nd Annual ACM/IEEE Int. Symp. on Microarchitecture*, pages 248–259, Nov. 1999.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb. 2002.

[4] T. Austin, D. Pnevmatikatos, and G. Sohi. Streamlining data cache access with fast address calculation. In *Proc. 22nd Annual Int. Symp. on Computer Architecture*, pages 369–380, June 1995.

[5] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embedded Computing Systems*, 1(1):6–26, Nov. 2002.

[6] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki. Instruction buffering to reduce power in processors for signal processing. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, 5(4):417–424, Dec. 1997.

[7] J. D. Balfour. *Efficient embedded computing.* PhD thesis, Stanford University, 2012.

[8] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, 8(3):317–326, June 2000.

[9] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 329–338, June 1988.

[10] P. Carazo, R. Apolloni, F. Castro, D. Chaver, L. Pinuel, and F. Tirado. L1 data cache power reduction using a forwarding predictor. In *Integrated Circuit and System Design, Power and Timing Modeling, Optimization, and Simulation*, volume 6448 of *Lecture Notes in Computer Science*, pages 116–125. Sept. 2011.

[11] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27–32, July 2008.

[12] N. Duong, T. Kim, D. Zhao, and A. V. Veidenbaum. Revisiting level-0 caches in embedded processors. In *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, pages 171–180, 2012.

[13] P. Greenhalgh. *Big.LITTLE Processing with ARM Cortex^{TM}-A15 & Cortex-A7 — Improving Energy Efficiency in High-Performance Mobile Platforms.* ARM Limited, Oct. 2011.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Int. Workshop on Workload Characterization*, pages 3–14, Dec. 2001.

[15] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proc. 37th Annual Int. Symp. on Computer Architecture*, pages 37–47, June 2010.

[16] S. Hines, D. Whalley, and G. Tyson. Guaranteeing hits to improve the efficiency of a small instruction cache. In *Proc. 40th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 433–444, Dec. 2007.

[17] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proc. Int. Symp. on Low Power Electronics and Design*, pages 273–275, Aug. 1999.

[18] B. Jeff. *Enabling Mobile Innovation with the Cortex^{TM}-A7 Processor.* ARM Limited, Oct. 2011.

[19] L. Jin and S. Cho. Macro data load: An efficient mechanism for enhancing loaded data reuse. *IEEE Trans. on Computers*, 60(4):526–537, Apr. 2011.

[20] N. P. Jouppi. Cache write policies and performance. In *Proc. 20th Annual Int. Symp. on Computer Architecture*, pages 191–201, 1993.

[21] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proc. 30th Annual ACM/IEEE Int. Symp. on Microarchitecture*, pages 184–193, Dec. 1997.

[22] H.-S. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson. Stack value file: Custom microarchitecture for the stack. In *Proc. Int. Symp. on High-Performance Computer Architecture*, pages 5–14, Jan. 2001.

[23] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero. Fast speculative address generation and way caching for reducing L1 data cache energy. In *Proc. Int. Conf. on Computer Design*, pages 101–107, Oct. 2006.

[24] NVIDIA Corporation. *Variable SMP (4-PLUS-1^{TM}) — A Multi-Core CPU Architecture for Low Power and High Performance*, 2011.

[25] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Design Automation of Electronic Systems*, 5(3):682–704, July 2000.

[26] C.-L. Su and A. Despain. Cache design trade-offs for power and performance optimization: A case study. In *Proc. Int. Symp. on Low Power Design*, pages 63–68, Apr. 1995.

[27] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proc. 35th Annual Int. Symp. on Computer Architecture*, pages 51–62, 2008.

[28] T. Weiyu, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *Proc. Int. Conf. on Computer Design*, pages 68–73, Sept. 2001.