Reducing Set-Associative L1 Data Cache Energy by Early Load Data Dependence Detection (ELD^3)

Alen Bardizbanyan[†], Magnus Själander[‡], David Whalley[‡], and Per Larsson-Edefors[†]
[†]Chalmers University of Technology, Gothenburg, Sweden
[‡]Florida State University, Tallahassee, USA
alenb@chalmers.se, msjaelander@fsu.edu, whalley@cs.fsu.edu, perla@chalmers.se

***Abstract*—Fast set-associative level-one data caches (L1 DCs) access all ways in parallel during load operations for reduced access latency. This is required in order to resolve data dependencies as early as possible in the pipeline, which otherwise would suffer from stall cycles. A significant amount of energy is wasted due to this fast access, since the data can only reside in one of the ways. While it is possible to reduce L1 DC energy usage by accessing the tag and data memories sequentially, thereby activating only one data way on a tag match, this approach significantly increases execution time due to an increased number of stall cycles. We propose an early load data dependency detection (ELD^3) technique for in-order pipelines that can detect if the load has a data dependency with a subsequent instruction that will cause stall cycles. If there is no such dependency, then the tag and data accesses for the load are sequentially performed so that only the data way in which the data resides is accessed. If there is a dependency, then the tag and data arrays are accessed in parallel to avoid introducing additional stall cycles. For the MiBench benchmark suite, the ELD^3 technique enables about 49% of all load operations to access the L1 DC sequentially. In a 65-nm physical implementation using commercial SRAM blocks, the proposed technique reduces L1 DC energy by 13%.**

I. Introduction

Energy-efficient computing is of critical importance today for mobile devices that run on batteries, for general-purpose processors that are performance limited due to processors hitting the power wall, and for data centers due to the rising cost of electricity and environmental concerns. Thus, some common computer design techniques need to be reexamined in the context of a greater emphasis on energy efficiency.

A level-one data cache (L1 DC) can have a significant effect on processor efficiency as it is accessed for every load and store operation. An L1 DC is often designed with a set-associative organization to lower its miss rate. These caches access both the ways of the tag and data arrays in parallel for loads in order to reduce the access latency so that a subsequent instruction using the loaded value is not stalled. However, a significant amount of energy is wasted since the desired data item can reside in at most one of the ways. A more energy-efficient method would be to access the tags first and then only access the single data way for the corresponding tag match. However, this method introduces stall cycles that hamper performance. As we show in Sec. IV, the execution time overhead of accessing tags and data sequentially is 8% on average in an in-order pipeline. These extra cycles also cause additional energy overhead due to more switching of the clock, etc., which can severely reduce the efficiency of the method.

In this paper we propose ELD^3, a technique that determines if the load has data dependencies with the next few instructions that will enter the pipeline. If no dependency is found, then the tags and data are sequentially accessed with no performance penalty. As we show in Sec. VI, this technique enables about 49% of the load operations to access a single way of data and, thus, reduces the L1 DC energy usage by 13% at a low execution time overhead of 0.6%.

II. Related Work

It is common for caches at higher levels of the memory hierarchy, e.g., level-two (L2) or level-three (L3), to access their tag and data arrays sequentially to save energy [1]. This approach is practical because memory accesses to the L2 and L3 caches are relatively few compared to L1 cache accesses, hence the execution time overhead is low. It has been shown that a sequential access approach for load operations to the L1 DC incurs an unacceptable performance penalty [2]. We propose a technique to selectively access the L1 DC sequentially for those load operations where we know beforehand that the longer access time will not cause an additional stall cycle, making sequential accesses to the L1 DC practical.

Many different techniques have been proposed to reduce energy in set-associative L1 DCs. These techniques mainly try to predict [2], [3] or retrieve [4] the way information before the L1 DC access so that only a single way of the L1 DC is accessed. The way-prediction techniques have a relatively high performance penalty of several percent [2], [3] compared to the proposed ELD^3 technique. Retrieving the way information requires a fully-associative search, with a speculative address in the address generation stage [4], which has a high overhead. In addition, the way information can only be retrieved if the speculation succeeds and if there is a hit in the fully-associative structure. As a result, the load operations that can benefit from this approach will be close to the proposed ELD^3 technique. Speculative tag access allows many load operations to access L1 DC sequentially by making the tag match speculatively in an early stage of the pipeline [5].But this technique can only be applied to virtually indexed physically tagged (VIPT) L1 DCs.

III. L1 DC Load Pipeline

A 3-stage access is common for the L1 DC in contemporary state-of-the-art in-order processors [1], [6], [7]. Fig. 1 shows how an N-way set-associative L1 DC access is performed, with respect to load operations that span three pipeline stages. During the first stage, the virtual memory address is generated for the L1 DC by adding an offset to a base address. In the second stage, the data translation lookaside buffer (DTLB), the tag, and the data memories are accessed and the tag match operation takes place. In the third stage, the way selection is done as a result of the tag hit signal, the retrieved data is formatted for halfword- and byte-level operations, and, finally, the data is forwarded to the execution units. The described memory access is for VIPT L1 DCs, which are commonly used by virtue of the reduced latency resulting from the parallel L1 DC and tag access that is inherent in a VIPT cache.

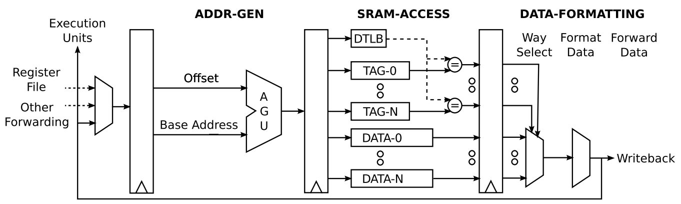

Fig. 1. L1 DC access that spans three pipeline stages.

While a VIPT cache reduces latency, ARM processors are mainly based on physically-indexed physically-tagged (PIPT) L1 DCs. For a PIPT organization, a very small DTLB structure is typically accessed directly after the address generation unit (AGU) in the address generation stage. This does not change how the tag and data arrays are accessed in the SRAM-ACCESS stage. Thus, the ELD^3 technique that we propose in Sec. V-B can be applied to both VIPT and PIPT L1 DCs.

IV. Data Dependencies

The latency of a load instruction depends on the number of pipeline stages needed to get the required data word from the L1 DC; it is two cycles for the 3-stage L1 DC pipeline described in Sec. III. When the load instruction reaches the end of the third stage, the data can be forwarded to the upcoming instruction. If there is a data dependency between a load instruction and the first or second succeeding instruction, then the pipeline of an in-order processor needs to stall two or one cycles, respectively, until the data value can be forwarded to the consuming instruction.

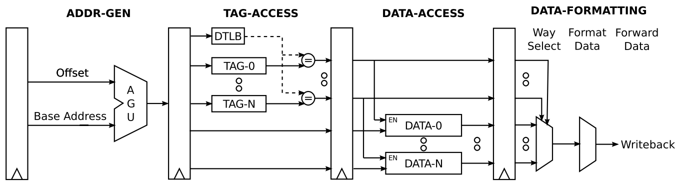

Fig. 2. L1 DC load access that spans four pipeline stages.

Fig. 2 shows an L1 DC pipeline in which the tag and data memory accesses happen in two consecutive stages. In this scheme, only one data way is accessed as determined by the tag hit signal. However, the scheme has a serious disadvantage: The sequential access increases the L1 DC access latency to three cycles and, thus, introduces many extra stall cycles compared to a 3-stage L1 DC access. There will be an additional stall cycle if there is a data dependency with any of the succeeding three instructions after the load.

Overall, across 20 MiBench benchmarks [8], about 50% of the load instructions have a data dependency with one of the upcoming three instructions. These are distributed as 19%, 12.8%, and 18.2% for the instruction in the next cycle, two cycles later, and three cycles later, respectively. Fig. 3 shows the execution time increase for an in-order processor (see Sec. VI) whose L1 DC access is increased from three to four stages to facilitate sequential tag and data accesses. Since data dependencies are so common, saving dynamic energy by making all L1 DC accesses sequential instead of parallel is not a feasible approach as it significantly degrades performance.

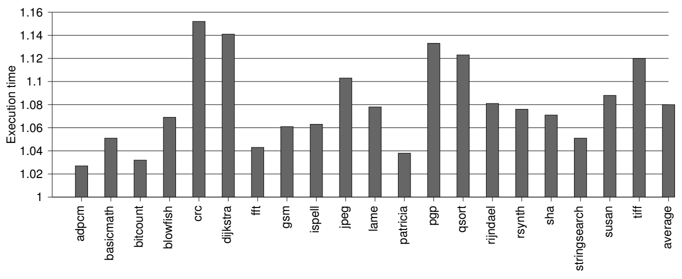

Fig. 3. Execution time overhead when accessing tags and data sequentially.

V. Early Load Data Dependency Detection to Enable Sequential Access

As explained in the previous section, parallel L1 DC tag and data access for load operations is essential to avoid stall cycles. However, this parallel access leads to a waste of energy in the event the load operation does not have a data dependency that causes a stall cycle.

Now assume that the data dependency information is known at the time of the load operation. It would then be possible to reduce the load energy for set-associative L1 DCs without inflicting any performance penalties. If there is a data dependency, the tag and data arrays can be conventionally accessed in parallel. If there is no data dependency, an extra pipeline stage can be used to sequentially perform the tag and data array accesses over two consecutive cycles. The tag hit signal can then be used to activate only one data way.

A. Dynamic Data Dependency Detection

To implement the concept above, the data dependency information must be available before the end of the address generation stage in order to mask the chip select signals of the data SRAM blocks. To reduce instruction decoding complexity, instruction sets are usually designed in such a way that the position of destination and source registers in the instruction word is always fixed. As a result, it seems that it would be possible for an in-order pipeline to compare the destination register of the load instruction with the instructions that enter the pipeline after the load instruction. However, this straightforward solution is not likely to be feasible.

Processors commonly have at least one instruction decode stage in which the instruction word is available for decoding and operations such as register file read. While the load instruction is in the address generation stage, the upcoming instruction word will be available in the instruction decode stage. This means that it is possible to check the data dependency

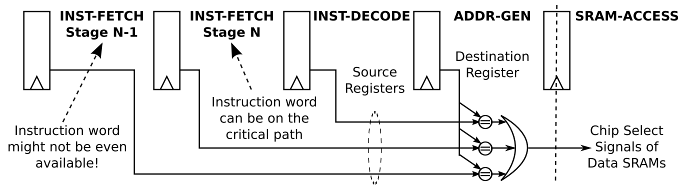
INST-FETCH Stage N-1
INST-FETCH Stage N
INST-DECODE
ADDR-GEN
SRAM-ACCESS
Source Registers
Destination Register
Instruction word can be on the critical path
Instruction word might not be even available!
Chip Select Signals of Data SRAMs

Fig. 4. Dynamic data dependency detection and its problems.

between a load and the instruction that immediately follows it. But checking the dependency between the load and the second and third upcoming instructions is not as straightforward. The second instruction following the load will be two stages behind the address generation stage and, thus, the instruction word might be available in the final part in this stage. As shown in Fig. 4, implementing a comparison with the instruction word in this stage is likely to either increase the critical path delay or constrain the paths in which the instruction word becomes available, leading to a significantly higher power dissipation for every instruction fetch. The third instruction after the load will likely not even be available when the load instruction is in the address generation stage.

B. Table-Based Data Dependency Detection

We propose a practical approach that provides early detection of a dependency on a load for most of the executed load instructions. We introduce an extra bit associated with each instruction word in the L1 instruction cache (IC) to indicate if the instruction is a load operation and if there is a dependence with one of the following three instructions. When a load operation is committed, a check is performed to identify if there is a dependency between the load and the subsequent three instructions. If a dependency is detected, then the corresponding bit is set to indicate that the tag and data ways are to be accessed in parallel to avoid an additional stall cycle. If no dependency is detected, then the corresponding bit is cleared and the next time the load operation is executed the tag and data SRAMs will be accessed in sequence across two consecutive stages. As long as the cache line with the load instruction is not evicted from the L1 IC, the bit will provide useful information regarding the data dependency and can be used for selecting either a parallel or sequential L1 DC access. When a cache line is evicted from the L1 IC the data dependency information might be incorrect. But the processor will still behave correctly even if the dependency information is incorrect, since in-order pipelines inherently have data dependency checks. Hence, if a load instruction is treated as it has no dependency when in fact it has, then it will only cause an additional stall cycle. Cache line evictions rarely happen and the dependency information if incorrect is updated after the first execution of a new load operation. The performance impact of the additional stall cycles caused due to incorrect dependency information is therefore insignificant. In addition, not having to set the data dependency bits to a specific state on a cache line eviction simplifies the implementation as access to all bits for a whole cache line is not required.

The extra bits could be directly included in the data SRAMs of the L1 IC. This solution presents a very low overhead in terms of bits, since the technique entails adding only one extra bit per word in the SRAM. However, this solution will complicate the design of the L1 IC, since the data dependency bits need to be updated independently of the instruction word. A more practical approach is to access a separate data dependency bit (DDB) memory in the address generation stage, decoupled from the instruction cache entirely. Moreover, since the operation of an instruction is already detected before the address generation stage, the DDB memory will only be accessed if there is a load operation. In contrast, if the bits are stored in the data SRAMs of the L1 IC, then all instructions will unnecessarily read out the extra bits.

The data dependency information is tracked by mainly using the logic of the conventional data dependency checks of an in-order pipeline. If the pipeline is stalled due to a data dependency, then the instruction causing the stall is marked with a bit that is passed through the pipeline together with that instruction. In addition, the bit read from the DDB memory for load operations is also kept with the load instruction throughout the pipeline. When the load instruction is to be committed, the two bits are compared and if they differ, then the DDB memory is updated.

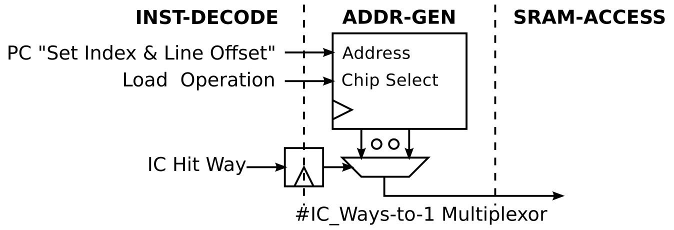

Fig. 5. The data dependency bit (DDB) memory access in the address generation stage.

Fig. 5 shows the proposed technique as part of the address generation stage. The bitwidth of the DDB memory is equal to the number of ways in the L1 IC. The height of the DDB memory is equal to the number of sets in one L1 IC way times the number of instruction words per cache line. The final multiplexer is a very small structure; it selects the correct bit, since for a given set and line index, the dependency bits from each way are read. Since the hit-way information is available at the beginning of the address generation stage, the critical path after the DDB memory read is going through the data-to-output path of the multiplexer, which is usually much faster than the select-to-output path due to less fanout. In addition, the bitwidth of the memory is much smaller compared to one data way of the L1 IC, which makes it faster.

VI. Results

We use extracted energy estimates from a placed and routed RTL description of a 5-stage in-order processor including 16kB 4-way set-associative instruction and data caches [9] with 32B line size [5]. Although the implemented pipeline is simpler than the pipeline evaluated in this work, the energy for accessing the L1 DC is representative. The RTL implementation is synthesized in Synopsys Design Compiler using a commercial 65-nm low-power process technology with multi-V_T standard cells and SRAM blocks. The layout work is done in Cadence Encounter. Energy is extracted using Synopsys PrimeTime PX on the RC-extracted netlist.

Table I shows the energy for different components of the L1 DC. The load and store operations include energy dissipated in the L1 DC peripheral circuits. The store energy is substantially smaller than the load energy due to store operations accessing only one data way. The energy for a 4-way data read is 106 pJ. When a load operation accesses the L1 DC sequentially, the energy of reading data from three ways is avoided, which results in a reduction of 79.5 pJ. For each load operation the overhead of accessing the DDB memory is also evaluated. In addition, the energy overhead of updating the DDB memory is also included in the final energy estimation. The leakage power is negligible due to the low-power process technology.

TABLE I
4-WAY SET ASSOCIATIVE L1 DC ENERGY (PJ)

Component	Energy	Component	Energy
L1 DC load	182.1	L1 DC 4-way tag read	57.3
L1 DC store	103.3	L1 DC 4-way data read	106.0
DDB memory read	8.6	DDB memory write	8.9

We use 20 benchmarks from six different categories of the MiBench benchmark suite [8]. The benchmarks are compiled using the gcc compiler with the -O2 optimization flag. We use the SimpleScalar simulator with the PISA instruction set [10] to model a 7-stage in-order processor with a 3-stage L1 DC pipeline. The L1 instruction and data caches are configured to be 16kB and 4-way set-associative. The simulator calculates the activities (loads, stores, cycles etc.) for the MiBench benchmark suite until completion. The final energy values are then calculated from these activities and the extracted energy of the various components of the L1 DC.

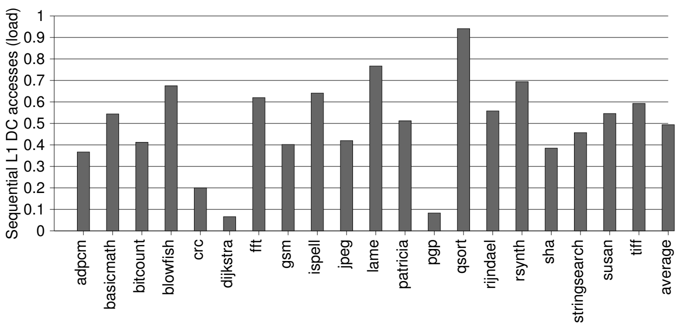

Fig. 6. Load operations that access only one L1 DC data way using ELD^3.

Fig. 6 shows how many load operations that sequentially access the L1 DC due to the ELD^3 technique and, thus, activate only one data way. On average, 49.4% of the load operations sequential access the L1 DC.

Fig. 7 shows the L1 DC energy dissipation for both stores and loads. The ELD^3 technique on average reduces the energy by about 13%. The energy reduction for loads has a strong correlation with the ratio of loads accessing a single L1 DC way of data. The energy for stores is unaffected. Note that for *dijkstra* and *pgp*, the energy dissipation increased slightly (1.8% and 0.9%, respectively) due to the overhead of checking the DDB memory outweighing the small ratio of loads that access only a single L1 DC way of data.

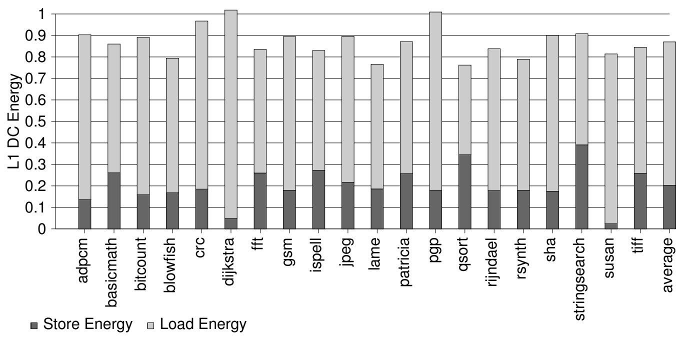

Fig. 7. L1 DC energy dissipation.

On average, 0.5% of the loads cause a stall cycle due to incorrect dependency information read from the DDB memory. In addition, 3.4% of loads cause a stall cycle due to a structural hazard in which a load that is sequentially accessed in the L1 DC is immediately followed by a load where the tags and data need to be accessed in parallel. This results in an 0.6% execution time increase.

VII. Conclusion

In this paper we show that early load data dependency detection (ELD^3) is an effective technique for reducing L1 DC energy dissipation with a very slight performance degradation. The DDB memory is accessed in the address generation pipeline stage only when a load instruction is detected during instruction decode. The ELD^3 technique can also be potentially used in combination with other approaches for making L1 DC accesses more energy efficient.

Acknowledgment

This research was supported in part by the the Swedish Research Council grant 2009-4566 and the US National Science Foundation grants CNS-0964413 and CNS-0915926.

References

[1] D. Williamson, *ARM Cortex A8: A High Performance Processor for Low Power Applications*, ARM.
[2] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proc. ACM/IEEE MICRO*, Dec. 2001, pp. 54–65.
[3] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proc. IEEE ISLPED*, Aug. 1999, pp. 273–275.
[4] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero, "Fast speculative address generation and way caching for reducing L1 data cache energy," in *Proc. IEEE ICCD*, Oct. 2006, pp. 101–107.
[5] A. Bardizbanyan, M. Själander, D. Whalley, and P. Larsson-Edefors, "Speculative tag access for reduced energy dissipation in set-associative L1 data caches," in *Proc. IEEE ICCD*, Oct. 2013, pp. 302–308.
[6] T. R. Halfhill, "ARM's midsize multiprocessor," *Microprocessor*, Oct. 2009.
[7] *MIPS® 1004K^{TM} Coherent Processing System Datasheet*, MIPS Technologies, Jul. 2009.
[8] 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*, Dec. 2001, pp. 3–14.
[9] V. Saljooghi, A. Bardizbanyan, M. Själander, and P. Larsson-Edefors, "Configurable RTL model for level-1 caches," in *Proc. IEEE NORCHIP*, Nov. 2012.
[10] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.