Tag Check Elision

Zhong Zheng, Zhiying Wang
State Key Laboratory of High Performance Computing & School of Computer
National University of Defense Technology
zheng_zhong@nudt.edu.cn, zywang@nudt.edu.cn

Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
mikko@engr.wisc.edu

ABSTRACT

For set-associative caches, accessing cache ways in parallel results in significant energy waste, as only one way contains the desired data. In this paper, we propose Tag Check Elision (TCE): a non-speculative approach for accessing set-associative caches without a tag check to save energy.

TCE can eliminate up to 86% of the tag checks (67% on average), without sacrificing any performance. These direct accesses to a 4-way set-associative data cache under TCE result in up to 56% and 85% data cache and Data Translation Look-aside Buffer (DTLB) dynamic energy saving, respectively.

Categories and Subject Descriptors

C.1.0 [**Processor Architecture**]: General; B.3.2 [**Memory Structure**]: Design Styles—*Cache memories*

1 Introduction

Caches have been playing an important role in efficiently bridging the speed gap between memory and CPU for decades. However, they consume a large fraction of the on-die area along with up to 45% of core power [19]. In addition, industrial sources and early research report that 3-17% of core power can be consumed by Translation Look-aside Buffer (TLB) [19, 9, 10]. In the many-core era, the power quota for each core is very limited, which requires simple core logic design along with more power efficient cache and TLB implementation without hurting performance.

Set-associative caches dominate the cache design in current commercial processors, as they provide higher hit rates, resulting in better performance. However, they require parallel way read and energy-hungry tag comparison. As a result, much of the energy is wasted on accessing bits that are discarded after tag check.

In this paper, we propose Tag Check Elision (TCE), a hardware approach to access set-associative caches without tag checks. TCE determines the correct cache way early in the pipeline by doing a simple bounds check that relies on the base register and offset associated with the memory instruction. In the same vein, the TLB access can also be eliminated in a physically-tagged cache. TCE memoizes the accessed cache ways and relies on a bounds check to decide if the later access is to the same line as an earlier access. The bounds check occurs as soon as the virtual address is available, and incurs no additional pipeline delay or performance degradation.

The results show that 35% to 86% (67% on average) of memory accesses in the SPEC CPU2006 benchmarks [6] can perform direct access to the cache. The cache dynamic energy saving is 15% to 56%, with 33% on average. The DTLB dynamic energy saving is 34% to 85% (66% on average). TCE outperforms two types of way prediction, including MRU and perfect prediction, in terms of Energy Delay Product (EDP).

The rest of the paper is organized as follows: The background and insight that motivate our work are presented in Section 2. The design of the TCE approach is described in Section 3. Experimental results are presented in Section 4. Related work is discussed in Section 5 and Section 6 concludes the paper.

ISLPED'14, August 11–13, 2014, La Jolla, CA, USA.

http://dx.doi.org/10.1145/2627369.2627606.

2 Motivation

For set-associative caches, all tags in a set must be checked to decide which way contains the requested data. To provide fast access in level one caches, all tags and data ways in a set are accessed in parallel, while only one way has the requested data. This redundant tag check and data access results in much energy waste. For example, in a 4-way set-associative cache, 4 tag checks and 4 data block reads are performed. In contrast, TCE reduces the cache access to just one data block read, eliding all tag checks and 3 other data block reads.

TCE is inspired by how memory addresses are generated. The example code in Figure 1(a) adds arrays b and c to form a new array a. The addresses to the array elements are computed by adding offsets to base registers, as shown in Figure 1(b)), where the base registers are rbp, $r13$ and $r12$, respectively. The index register rsi is incremented by 4 in each loop iteration, providing the offset for the array addresses.

The addresses for these arrays comprise static base registers and an increasing index register. As the cache line size (typically 64B) is much smaller than the physical page size (typically 4KB), the same cache line access can be determined just by comparing the virtual addresses. However, this virtual address comparison cannot be performed until address generation. Fortunately, if the base address register

```
for( i=0; i<N; i++ )
{
    a[i] = b[i] +c[i];
}
```

(a) Loop

```
1. mov   $0x0,%esi
   Loop:
2.   mov   0x0(%r13,%rsi,1),%edx
3.   add   (%r12,%rsi,1),%edx
4.   mov   %edx,0x0(%rbp,%rsi,1)
5.   add   $0x4,%rsi
6.   cmp   $0x10,%rsi
7.   jne   Loop
```

(b) Instructions in X86

Figure 1: Loop to add two arrays and its corresponding instructions in X86 ISA.

is unchanged (e.g. base registers shown in Figure 1(b)), the same cache line determination could be simplified to just compare the offsets (e.g. register rsi) with a bounds check. Based on this observation, we design a mechanism to determine same cache line accesses by keeping a cache line record for the base register and comparing the offset value. Once the same cache line access is detected, the access to the cache can be performed directly without any tag check or TLB access. This type of access elides tag checks, and thus we name this approach as Tag Check Elision. The sufficient condition for the same cache line determination is:

- The same base register value,
- And the offset is within the cache line bounds.

Overall, the advantages of our TCE approach are:

(a) The stored cache line information and the offset bounds are read the same time as performing register value read, as they are indexed by base register id instead of register value or address;

(b) The offset comparison is performed in parallel with the address generation, which does not add any delays to the critical data path;

(c) TCE completely elides the tag check for the determined same cache line access;

(d) On a direct access determined by TCE, the access to the TLB is bypassed to save more energy.

3 TCE design

3.1 Cache Way Memoization

To memoize the accessed cache lines, cache way records (CWRs) are added to the processor, as shown in Figure 2. The number of CWRs matches the number of architected fixed-point registers in the ISA, as one record is kept for one register that will be used as base address register. This CWR design allows it to be indexed by the register id and be accessed early, the same time as the register read in the pipeline.

Each CWR includes three fields, namely *valid*, *bound*, and *cache pointer*. The valid field indicates if the corresponding cache block information is valid or not. The bound field gives the offset range that is located within the current cache line with the same base register value. As the range length is exactly the cache line size, we can just keep the lower bound in the record to save space. The cache pointer records the last cache line accessed based on the current base register.

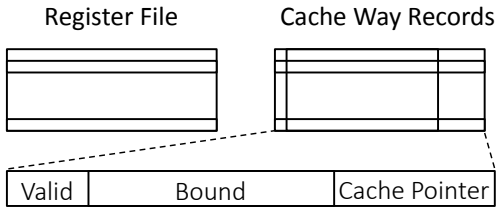

Figure 2: Example of cache way records organization.

3.2 Walkthrough Example

An example of how the TCE mechanism works is shown in Table 1. This example shows how the register *rdx* and its corresponding CWR changes.

At the beginning, we assume that the value of the register *rdx* is *0x40043c8* and the CWR for register *rdx* is invalid. The instruction in the first row accesses the memory whose address comprises the value in register *rdx* and an immediate number *4*. As the valid bit in the CWR is 0, the cache access is performed in the normal mode (probing all ways in parallel) and the cache way number (e.g. 3) of the current access is obtained to build the CWR. According to register *rdx*'s value and the current offset value, the bounds for the current accessed cache line are calculated, which are -8 and 56 (-8 + cache line size, 64). Thus, the CWR state for *rdx* becomes | 1 | -8 | 3 | after the first instruction.

When executing the second instruction, the CWR is valid and the offset 8 is within the bounds between -8 and 56. Direct access to the desired cache way is performed to fetch data. This case is what we expect, accessing cache directly and bypassing DTLB access to save dynamic energy.

On the third instruction, the offset 64 is out of bounds of the current CWR. In this situation, the cache access must be performed in normal mode, and the CWR is rebuilt after finishing cache access (assuming the accessed cache way number is 2). The CWR is invalidated after the last instruction as the value of register *rdx* is modified.

For simplicity of illustration, this example uses immediate values as offsets to describe our proposed approach. However, this approach works the same for the offsets that are stored in the register file, as shown in Figure 1.

3.3 TCE Enhancements

3.3.1 Multiple records.

Keeping one record for each register will possibly suffer from capacity misses. With multiple records, more cache line information can be kept. If we keep N records for one register, then we can track the last N cache line that have been accessed based on the same register value. However, keeping too many records will complicate the management of the CWRs.

In our design, initial experiments showed that provisioning *two* CWRs (Double Record, DR) for each register forms a reasonable compromise between complexity and performance.

3.3.2 Register value tracking.

As illustrated in Figure 3, some array accesses rely on just a single base register rax. The base register is incremented by 4 in each iteration, causing CWR rebuilding. To deal with the small changes to the register, another optimization is added to keep track of the register value to avoid unneces-

	Instruction	Action	Record state
1	Mov 4(%rdx), %rcx	**[Miss]:** Access the cache in normal mode, get the way pointer.	1 | -8 | 3
2	Add 8(%rdx), %rcx	**[Hit]:** The offset 8 is within the bounds between -8 and 56 (-8+64). Then access the cache block through direct access.	1 | -8 | 3
3	Mov %rcx, 64(%rdx)	**[Miss]:** The offset 64 is out of bounds between -8 and 56. Access the cache in normal mode and rebuild the record.	1 | 56 | 2
4	Mov %rax, %rdx	**[Invalidation]:** The value of the base register *rdx* has changed. The corresponding cache way record for register *rdx* must be invalidated.	0 | X | X

Table 1: An example to show how the cache way record works, using X86 ISA.

```
for (i = 0; i< N; i++)
{
   sum += a[i];
}
```

(a) Loop

```
1. lea   0x1000(%rsp),%rcx
   Loop:
2.    add   (%rax),%edx
3.    add   $0x4,%rax
4.    cmp   %rcx,%rax
5.    jne   Loop
```

(b) Instructions in X86

Figure 3: Loop to sum the array a and its corresponding instructions in X86 ISA.

Register Value	Valid	Bound	Cache Pointer

Figure 4: Example of backup buffer design for the CWRs.

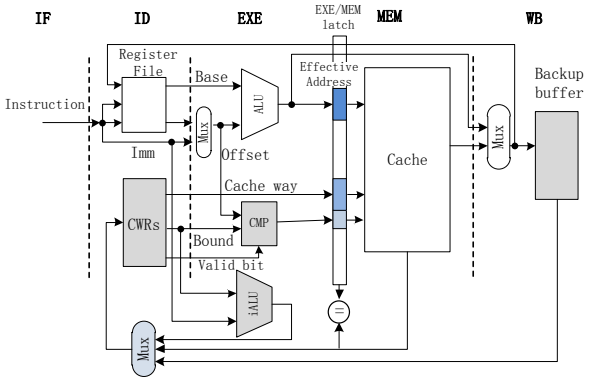

Figure 5: Pipeline integrated with TCE.

sary invalidation; this applies whenever the accesses are still located in the same cache line.

This optimization is called Register Value Tracking (RVT), which keeps the CWR valid by tracking the register value and adjusting the corresponding bound field. On the instruction *add $0x4, %rax* , we update the CWR by subtracting the bound with the same immediate $0x4$, instead of invalidating. Thus, the CWR still points to the same cache line and the next access has a chance to hit.

3.3.3 *Backup buffer.*

When the register value changes, the corresponding CWR entry must be invalidated. However, it is possible that this discarded CWR would be useful later. To prevent unnecessary CWR rebuilding, a backup buffer can be added to buffer the CWRs that are discarded because of the register value change.

As the CWR is strongly coupled with the register value and the offset, each item in the backup must contain the register value, as shown in Figure 4. The backup can be organized as an inexpensive direct-mapped cache. The tag is the register value, and the data is the corresponding CWR item.

3.3.4 *Energy modeling.*

The energy consumption of CWRs check, register value tracking, and backup buffer are carefully modeled and integrated into McPAT [12].

3.4 Coherence and Correctness

To guarantee correct access to the cache without tag check, the CWRs must be kept coherent with the cache lines. Once a cache line is evicted or invalidated, the CWR entries that point to this cache line must be invalidated. Instead of using energy-intensive associative lookups as [20], we keep backward pointers in each cache line.

For each cache line, we add a vector to indicate which register's CWRs have pointers pointing to this cache line, one bit for each register. When the cache line is accessed in normal mode to build a CWR, the vector bit for the CWR will be set. As the number of fixed-point registers is relatively small, the cost of the vector bits will be fairly low.

For a small number of backup buffer entries, the same bit vector mechanism can be extended to cover the backup buffer. We show in the next section that a 16-entry backup buffer is a good design choice, considering the overhead and complexity of keeping it coherent.

For other events that will possibly make CWR entries go stale, for example, memory map changes, page access permission changes, and thread context switches, TCE invalidates all of the CWRs. TCE also bypasses DTLB lookups; hence, any DTLB replacement will invalidate all CWRs to guarantee correctness.

3.5 TCE Design in the Pipeline

As illustrated in Figure 5, the structures added to implement the TCE are shaded in the pipeline. For simplicity, the pipeline components that are unaffected by TCE are not shown in the figure. The register-id-indexed CWR is accessed at the same time as the register file read, and the bounds check for the offset is finished in parallel with address generation, by a dedicated comparator *CMP*. The bounds check results and the cache way information are sent to the cache to decide which mode should be adopted to finish the data access. The iALU component is dedicated for RVT, and the backup buffer is accessed in the WB stage.

The direct access information is stored in EXE/MEM latch before reaching MEM stage. The cache way information can be bypassed to the EXE/MEM latch to satisfy back-to-back access, like the first and second instruction in Table 1. When cache lines get evicted, TCE also checks the

Parameters	Value
Processor	One core, X86 ISA, In-order
Base L1 DCache	32K, 4-way, 3 cycles, 64 B
L2 Cache	2MB, 16-way, 12 cycles, 64 B
DTLB	64 entries, fully-associative
Basic CWR	32 bits, 16 entries (32 for DR)
Backup buffer	96 bits, 16-1024 entries
DCache back pointer	16 bits per cache line, 1KB in total

Table 2: System configuration parameters.

EXE/MEM latch to invalidate any inflight matching TCE cache access.

4 Evaluation

4.1 System Configuration

The system configuration in our evaluation is listed in Table 2. Our base architecture is a single core X86 in-order processor. As tens to hundreds of cores are integrated into one chip, simple cores are appealing due to their area and power efficiency (e.g. Intel® Xeon Phi co-processor [5]). As TCE deals with L1 caches, it is scalable to multi-cores. TCE can also be adapted to out-of-order cores, but that is beyond the scope of this paper.

We use Gem5 [2] in Syscall Emulation mode (SE mode) for detailed architecture simulation, with Simpoints [18] for fast simulation. TCE mechanism and a simple DTLB simulation component are added to Gem5.

The test programs come from SPEC CPU2006 [6], with input data size *train*. The benchmarks are compiled and statically linked with GCC-4.7.2, G++-4.7.2 and gfortran-4.7.2. We integrated TCE component into McPAT [12] to estimate the whole processor energy dissipation.

The baseline systems that we compared with are two way prediction schemes, MRU and perfect prediction, where perfect prediction is not realistic and just for reference. As way prediction incurs performance degradation, we adopt Energy Delay Product (EDP) as metric for comparison.

4.2 CWR Hit Rate

The total CWR hit rates for SPEC benchmarks under different optimization are shown in Figure 6. We evaluate the effectiveness of the base design and optimizations by cumulatively adding optimizations and increasing the backup buffer size.

Generally, the hit rate improves as more optimizations are applied over the base design, and with more backup buffer entries. On average, the base design can capture 40.95% of the memory access, 36.96% and 43.77% for SPECint and SPECfp, respectively. The effectiveness of the basic design and optimizations vary widely among different benchmarks. The reason for this difference is two-fold: the data access pattern in the program and the binary code compiled and optimized by compilers. The data access pattern depends on the benchmark's algorithm and how it is implemented.

The most significant difference among the benchmarks comes with the RVT optimization. *Libquantum* has more than 50% increase, and another five benchmarks from SPECint and three from SPECfp have more than 20% increase. In contrast, *mcf*, *cactusADM* have just around 1% increase.

With DR, RVT and a 16-entry backup buffer, up to 86% of accesses (*hmmer*) can perform direct access, with 67% on average. When increasing the backup buffer entry number to 1024, the average hit rate improves by a marginal 4.7% over the 16-entry design, while incurring significant area and energy overhead.

4.3 Energy

4.3.1 Data cache and DTLB dynamic energy saving.

As shown in the last section, a large number of backup buffer entries offer marginal benefit while increasing complexity and energy consumption. The rest results are be based on **Base + DR + RVT + 16 BBs**.

The cache and DTLB dynamic energy saving for SPEC CPU2006 benchmarks are shown in Figure 7 (left). The energy savings for cache dynamic energy vary widely, with around 55.69% for *sphinx3* to just 15.01% for *gamess*. Generally, the proposed TCE mechanism reduces the cache and DTLB dynamic energy by 32.72% and 66.38% on average, respectively.

A higher hit rate in the CWR does not necessarily result in higher cache dynamic energy saving, because writes to the data cache will first check tag then write to the data array. TCE writes can only save the tag check energy, whereas TCE reads can save energy on both tag check and data array read. For example, *milc* enjoys higher total hit than *h264ref*, 84.09% and 79.44 %, respectively. However, the cache dynamic energy saving for these two benchmarks are 43.60% and 49.67%. The reason is that TCE reads in *h264ref* account for about 68% of the total memory access while in *milc* reads are around 60%, as shown in Figure 7 (right).

As the DTLB miss rate is quite low and the energy consumption for TLB lookup is much higher than for TLB replacement, the DTLB access energy accounts for around 99% percent of total DTLB energy. Thus, the percentage of dynamic DTLB energy saving is almost the same as the hit rate in the CWR.

4.3.2 Area and energy overhead.

The biggest area overhead comes from the back pointer vector, with 1KB capacity in total. However, the total area overhead is less than 0.1% of the total chip area. In addition, TCE design adopts small structures for CWR and backup buffer, compared with data cache. The energy cost to accessing these structures and doing optimization is around 0.5% of the data cache energy according to McPAT, on average.

4.4 Comparison with Way Prediction

In this section, the TCE approach is compared with the most closely-related prior work, way prediction, which similarly does not rely on the compiler or ISA modification.

Two way prediction schemes are included to give a better comparison: (a) Ideal Most Recent Used (iMRU): MRU technique predicts the last accessed way in a set to be the next. In this evaluation, we assume an ideal MRU, where no delays will be added to the critical path and no energy will be consumed to access the predict bit before cache access. (b) Perfect Prediction (PP): The PP scheme can correctly predict every access without any cost (area or energy cost), which *ONLY* misses on data cache miss. This prediction cannot be achieved by any prediction scheme. This is included *ONLY* as a reference to the best-case potential of all previously-proposed way prediction approaches.

Energy delay product (EDP) is reported to compare way prediction with the TCE approach.

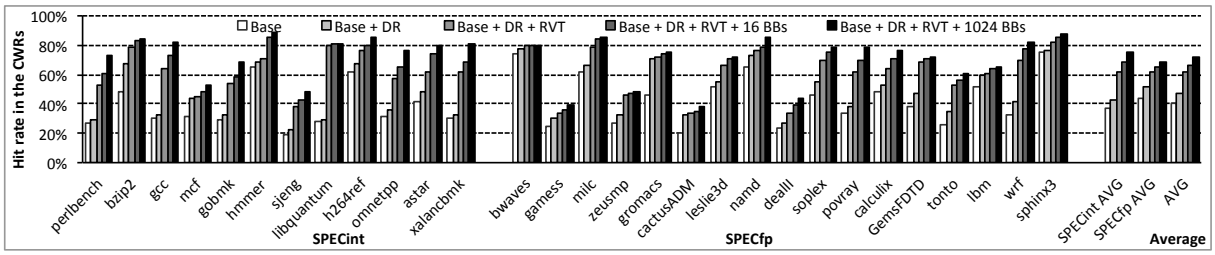

Figure 6: CWR hit rate for data cache access under different optimizations.

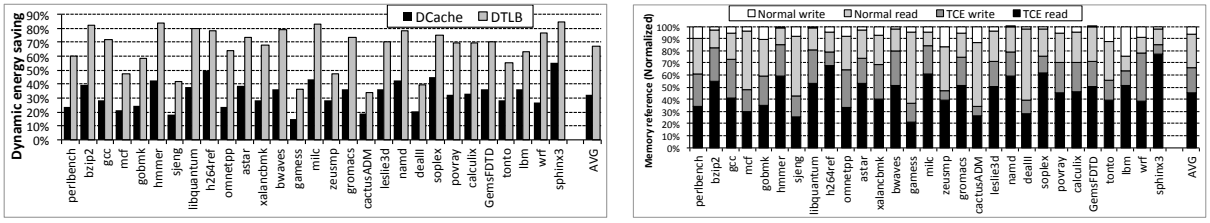

Figure 7: Data cache and DTLB dynamic energy saving (left), and TCE access breakdown (right).

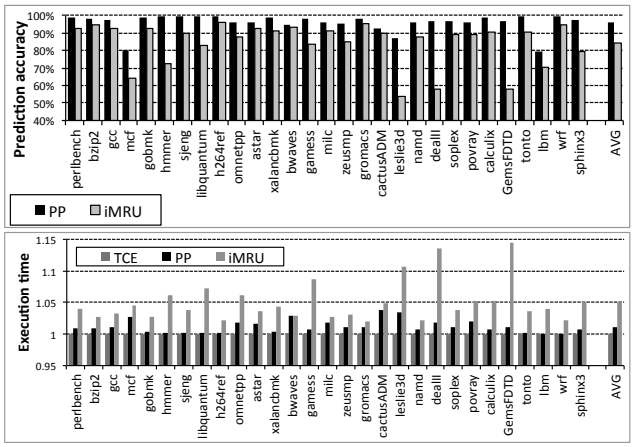

Figure 8: Prediction accuracy for PP and iMRU (top) and execution time for TCE, PP, and iMRU (bottom).

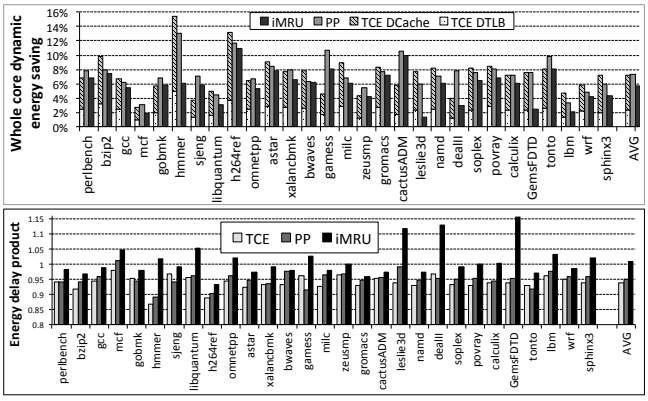

Figure 9: Whole core dynamic energy saving (top, higher is better) and energy delay product (bottom, lower is better) for TCE, PP, and iMRU.

4.4.1 *Performance.*

The prediction accuracy and execution time under PP and iMRU schemes are shown in Figure 8, respectively. The performance of TCE is also included, but it always the same as the baseline. As shown in the Figure 8 (top), the prediction accuracy of PP is quite high for most of the benchmarks, as it *ONLY* misses on data cache miss. The prediction accuracy for the more realistic approach, iMRU, is lower than PP, with 12% difference. Way prediction requires re-accessing the cache way on a wrong prediction, incurring performance degradation. The average performance degradations (shown in Figure 8 (bottom)) are about 1.18% and 5.02% more execution time for PP and iMRU, respectively. And the worst cases are 3.78% and 14.57% for PP and iMRU, respectively.

4.4.2 *Energy delay product (EDP).*

The whole core dynamic energy saving and corresponding EDP are illustrated in Figure 9. The variation of these results comes from two aspects: (a) differences in cache dynamic energy saving for TCE and way prediction on different benchmarks; (b) cache dynamic energy accounts for different fractions of the whole core dynamic energy for different benchmarks.

TCE dynamic energy saving comes from cache dynamic energy saving and DTLB dynamic energy saving. In contrast, way prediction (PP and iMRU) only saves dynamic energy in the cache. Generally, the average whole core energy saving for TCE is almost the same as PP, better than iMRU. However, PP is not realistic, and TCE significantly outperforms iMRU across benchmarks.

On average, TCE outperforms both PP and iMRU in terms of EDP. Because of the delay caused by misprediction, TCE outperforms iMRU on *ALL* cases. TCE has significant advantages over iMRU on several benchmarks, for example, *mcf*, *hmmer*, *libquantum*, *leslie3d*, *dealII*, *GemsFDTD*. These big differences come from the misprediction penalty for longer execution time.

The most significant advantage of TCE is that it reduces the cache dynamic energy without hurting performance across all the benchmarks. On the contrary, EDP for way predic-

tion can be higher than the baseline system, such as *libquantum* and *GemsFDTD*.

5 Related work

The work most similar to TCE to reduce the cache way access is way prediction. Predictive sequential associative cache [3] uses a number of prediction sources to pick the first block in a set to probe. On a miss to the predicted way, the other ways are checked. Similar way prediction techniques have been proposed in [8] and [1]. Powell et al. [16] combined way prediction and selective direct-mapping to reduce L1 dynamic cache energy. Besides performance penalty on misprediction, physical address must be obtained from TLB to check if the prediction is correct.

Instead of prediction, way caching [15] [14] records the way of recently accessed cache lines to reduce the dynamic energy of highly associative caches. A problem for this technique is that the way cache is accessed before cache access, which will add delay to the data access in the critical path. A similar approach, way memoization [13], adds the lower 14 bits of base address and displacement to index the memoized recent accessed cache ways. However, it is designed for application specific integrated processors and assumes that the 14-bit add and structure indexing can be finished in parallel with 32-bit add, which does not adapt easily to general purpose processor.

Tagless cache [17] restructures the first-level cache and TLB for more efficient storage of tags, achieving substantial energy gains, but, unlike TCE, does not elide TLB accesses, requires changes to the replacement policy, affects miss rates and performance in unpredictable ways, and complicates support for coherence and virtual address synonyms.

Before accessing the cache, techniques are proposed to filter unnecessary accesses. Sentry tag [4] determines the mismatched ways and halts them to save energy. The way halting technique [21] is an extension of the concept of sentry tags. Way decay [11] and way guard [7] adopt a bloom filter to reduce the ways that need to be checked. One problem for this type of techniques is that a new structure must be accessed serially, after address generation before accessing the cache, hence either increasing cycle time or adding a pipestage to the memory access latency.

6 Conclusion

To reduce set-associative cache access energy, we propose Tag Check Elision, which avoids tag checks and TLB access without causing performance degradation. TCE memoizes the accessed cache line and the offset of the memory address for a base address. Access to the same line elides tag checks and TLB lookups by comparing the offset of the memory address that use the same base register. To improve the base design, three optimizations are proposed, namely double records, register value tracking and a backup buffer.

We evaluated the effectiveness of the base TCE approach and further optimizations on X86 for reducing L1 data cache energy. The TCE approach avoids 35% to 86% of tag checks, which results in data cache energy savings of 15% to 56%, and DTLB dynamic energy saving of 34% to 85%. Under the TCE approach, set-associative caches achieve better energy delay product than way prediction.

7 Acknowledgments

We thank the anonymous reviewers for their insightful feedback, which has improved the content and presentation of this paper. This work is partially supported by CSC, China's 863 Program (No. 2012AA010905), NSFC (No. 61070037, 61272143, 61272144, 61103016, 61202121), NUDT's innovation fund (No. B120607), and RFDP (No.20114307120013). This work is also supported in part by NSF grant CCF-1318298.

8 References

[1] B. Batson and T. N. Vijaykumar. Reactive-associative caches. In *PACT '01*, pages 49–60, 2001.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH CAN*, 39(2):1–7, Aug. 2011.

[3] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *HPCA '96*, pages 244–253, 1996.

[4] Y.-J. Chang, S.-J. Ruan, and F. Lai. Sentry tag: an efficient filter scheme for low power cache. In *CRPIT '02*, pages 135–140, 2002.

[5] G. Chrysos and S. P. Engineer. Intel® xeon phi coprocessor (codename knights corner). 2012.

[6] S. P. E. Corporation. SPEC CPU2006 Site, 2013.

[7] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee. Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In *ISLPED '09*, pages 165–170, 2009.

[8] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED '99*, pages 273–275, 1999.

[9] T. Juan, T. Lang, and J. J. Navarro. Reducing tlb power requirements. In *ISLPED '97*, pages 196–201, 1997.

[10] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating physical addresses directly for saving instruction tlb energy. In *MICRO-35*, pages 185–196, 2002.

[11] G. Keramidas, P. Xekalakis, and S. Kaxiras. Applying decay to reduce dynamic power in set-associative caches. In *HiPEAC'07*, pages 38–53, 2007.

[12] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO-42*, pages 469–480, 2009.

[13] A. Ma, M. Zhang, and K. Asanovic. Way memoization to reduce fetch energy in instruction caches. In *ISCA Workshop on Complexity Effective Design*, page 31, 2001.

[14] R. Min, W.-B. Jone, and Y. Hu. Location cache: a low-power l2 cache system. In *ISLPED '04*, pages 120–125, 2004.

[15] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. Reducing power consumption for high-associativity data caches in embedded processors. In *DATE '03*, pages 1064–1068, 2003.

[16] 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 *MICRO-34*, pages 54–65, 2001.

[17] A. Sembrant, E. Hagersten, and D. Black-Shaffer. Tlc: A tag-less cache for reducing dynamic first level cache energy. In *MICRO-46*, pages 49–61, 2013.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS X*, pages 45–57, 2002.

[19] A. Sodani. Race to exascale: Opportunities and challenges. In *MICRO 2011 Keynote talk.*, 2011.

[20] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *MICRO-34*, pages 124–133, 2001.

[21] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. *ACM TACO*, 2(1):34–54, 2005.