RegionTracker: Using Dual-Grain Tracking for Energy Efficient Cache Lookup

Jason Zebchuk and Andreas Moshovos
Department of Electrical and Computer Engineering
University of Toronto
{zebchuk, moshovos}@eecg.toronto.edu

Abstract

This work proposes energy efficient memory hierarchy lookup structures aimed primarily at relatively large, higher-level on-chip caches. The mechanisms proposed provide location information for a large fraction of cache references and eliminate the corresponding accesses to a larger, slower and less energy efficient tag array. A key contribution of this work is the concept of dual-grain tracking *where a two-level, two-grain approach is used to dynamically focus a set of few tracking resources on high-payoff memory areas. A coarse-grain tracking structure uses imprecise information to identify accesses to new regions of memory and then directs the allocation of a precise, fine-grain tracking structure. We propose* RegionTracker, *a simple implementation of dual-grain tracking which can be easily partitioned for optimizing its power and latency, and which does not use cascaded lookups or impose any restrictions on cache placement. We demonstrate that RegionTracker can significantly reduce lookup energy for various L2 caches. For example, we show that a RegionTracker that uses just 6.9% of the storage used by a conventional tag array and that can track just 128 8Kbyte regions, is able to reduce L2 lookup energy by 35% on the average for a 4MB L2 cache. We also demonstrate that RegionTracker can complement conventional, demand-driven tag set buffers and that it provides better energy savings.*

1 Introduction

This work proposes simple mechanisms to reduce cache lookup energy in higher level (L2 and L3) on-chip caches while targeting applications with relatively large memory footprints. A number of application, semiconductor technology and microarchitectural trends suggest that the contribution of tag lookup power to overall processor power will increase in the future. Tag energy will increase as higher level caches become larger and are accessed more frequently.

The size of higher level caches will increase as a result of application and semiconductor technology trends. Historically, application memory footprints and working sets for "typical" applications have grown and evolved. At the same time, the gap between processor and memory speeds has also grown. Larger on-chip caches help reduce the combined effects of these two trends.

Other semiconductor and microarchitecture trends will result in an increased number of accesses to higher level caches. Specifically, although semiconductor technology improvements have lead to smaller and faster transistors, corresponding increases in processing speeds have limited the amount of SRAM storage that can be accessed within a reasonable number of clock cycles. This combines with the low latency requirement of first level caches to limit the size of L1 caches. This limitation suggests that higher level caches will be accessed more often. Recent trends towards simultaneous and fine-grain multithreaded cores, as well as towards chip-multiprocessors have also resulted in larger high level, on-chip caches with increased traffic. Hardware and software prefetching further increase the demand for cache bandwidth. Finally, this increased cache traffic is becoming unbalanced as some requests, such as many coherence and prefetching requests, only access the tag arrays.

This works proposes *RegionTracker*, a complexity effective mechanism for increasing the energy efficiency of cache lookups. RegionTracker supplements the tag array, providing the same information for many lookups and thus eliminate man tag array accesses. RegionTracker uses two simple structures. The first structure tracks which coarse grain regions currently have blocks cached. It uses this information to detect the first access into newly touched regions. A second structure maintains fine-grain location information for individual blocks within regions (i.e., where the blocks are cached), but only for a small number of regions, as instructed by the coarse-grain tracking structure. As we explain in more detail in Section 2, typical application behavior is such that these two structures can be used to locate the blocks referenced by many cache accesses, exploiting the same behavior that makes small translation look-aside buffers effective.

As Section 3 explains, the implementation of RegionTracker is straightforward and imposes no additional restrictions on what can be cached simultaneously; it also requires no changes to existing cache implementations and avoids associative lookups

and updates. Imprecise information is used by the coarse-grain tracking structure, allowing a simple implementation at the price of capturing most relevant requests but not all. Key to the energy efficiency of RegionTracker is that each access precisely addresses a very small portion of the RegionTracker structures. Although RegionTracker can also potentially reduce lookup latency and improve performance, this work focuses on RegionTracker's ability to increase lookup energy efficiency.

RegionTracker is an example of mechanisms that rely on the concept of *dual-grain tracking*, or DGT for short. DGT mechanisms track block residency information at two levels of granularity so that a relatively small structure can efficiently satisfy many cache lookups.

This work makes the following contributions: (1) it introduces the concept of DGT; (2) it proposes *RegionTracker*, an energy efficient implementation of DGT, and demonstrates that practically sized RegionTrackers can reduce energy significantly (e.g., 35% of lookup energy for a 4Mbyte cache with a RegionTracker that requires just 6.9% of the resources required by a conventional tag array); (3) finally, it shows that RegionTracker provides better energy reduction than tag set buffers.

The rest of this paper is organized as follows: In Section 2 we introduce the DGT concept and briefly discuss how lookup energy can be reduced. Section 3 presents the RegionTracker implementation. We review related work in Section 4. In Section 5 we demonstrate RegionTracker's utility, and compare it to an existing technique for tag lookup energy reduction. Finally, in Section 6 we summarize this work.

For clarity, and without the loss of generality, we will use the term *tags* for conventional memory hierarchy lookup structures. The techniques we discuss, however, are applicable to other recently proposed lookup structures such as the centralized lookup arrays of the NuRapid memory hierarchy [10]. We also restrict our attention to level-two caches, however, the methods proposed should be directly applicable to even higher cache levels. Finally, all L2 caches used in this study are 8-way set-associative because through experimentation we found that our techniques are not noticeably sensitive to associativity. We also assume that the L2 uses 128-byte blocks (a commonly used size today).

2 Dual-Grain Tracking

RegionTracker (the implementation details are given in Section 3) achieves high energy efficiency via a two-level, dual-grain tracking (DGT) approach where the first level uses coarse-grain tracking and the second level uses fine-grain tracking for only a few, large memory regions. A *region* is a large continuous, aligned memory area of power of two size.

The coarse-grain level aims at detecting newly touched regions that have no blocks currently cached. It does so by detecting *first misses*. An access for block B within region R sent to cache C is a first miss if and only if no block within region R, including B, is currently cached within C. Once a first miss is detected, the fine-grain tracking level starts tracking the location of all blocks within the region. This is done by recording whether or not a block is cached, and if so, in which data way it is cached. This requires only a few bits per block, as opposed to a full tag. It is important to observe that when a first miss is detected, complete location information is also detected for the whole region since none of its blocks are currently cached. Thus, a single access uncovers information for many blocks, allowing the fine-grain tracking level to track all blocks as they accessed. This property eliminates the need for an initial search of the L2, and makes DGT effective despite a lack of substantial temporal locality in the L2 stream (temporal locality is typically absorbed by the L1 cache).

RegionTracker was designed primarily to exploit a behavior that is typical of many applications with large memory footprints. Specifically, although these applications access a very large set of regions over their lifetime, they typically operate on a few memory regions at any given time. The first time an application accesses a region, it incurs a first miss, giving RegionTracker an opportunity to track subsequent references within that region. Assuming that only a few regions are accessed at a time, RegionTracker should be able to track them all successfully using few resources. Much later, after many regions have been touched, a region may be accessed again. Given that the application has a large memory footprint, it is likely that all previously accessed blocks within the region have since been evicted from the cache as a result of capacity and, to a lesser extent, conflict misses. Accordingly, another first miss will occur and RegionTracker again has the opportunity to detect it and start tracking the region.

2.1 Reducing Lookup Energy with DGT

RegionTracker impacts energy and latency as summarized in Table 1. Prior to accessing the tag array for each cache lookup, RegionTracker is examined. Ideally, RegionTracker provides sufficient information to completely avoid the tag access. This assumes an in-series tag and data array organization for the L2 where the RegionTracker is accessed first and then the tags are accessed only if needed, and finally a single data way is accessed. This is the norm in commercial designs because it reduces power, e.g., [6]. We define a *hit* in RegionTracker as an access which indicates definitively

that the requested block is either not in the cache, or is located in a specific cache way. In the first case, only a single way of the tag array needs to be accessed in addition to replacement tracking information to determine and update the tag that will be replaced. In the latter case, there is no need to access the tag array at all. A lookup *miss* occurs when RegionTracker provides no precise information. In this case, more energy is used as the tag array has to be accessed after the RegionTracker.

Table 1. How RegionTracker (DGT column) impacts power and latency and which parts of the conventional L2 tag array have to be accessed.

DGT	*L2*	*Energy*	*Latency*	*L2 Tag Access*
miss	*miss*	increased	increased	all ways
hit	*miss*	decreased	decreased	single way + replacement information
miss	*hit*	increased	increased	all ways
hit	*hit*	decreased	decreased	status bits for one way as needed

3 RegionTracker Design and Application

The RegionTracker implementation of DGT studied in this work consist of two structures: (1) the *Cached Region Hash* or *CRH,* and (2) the *Cached Block Vector,* or *CBV*. The CRH is used to detect the first miss into a region and is identical to the CRH proposed in [24]. The CBV tracks the location of all the blocks within the few regions that are currently fine-grain tracked. The organization of both structures is shown in Figure 1. Both structures are indexed using parts of the incoming address. Without loss of generality, in this section we assume a 2Mbyte L2 cache, 42-bit physical addresses and 8Kbyte regions. The relevant parts of the incoming address are a unique *region number* (bits 41 through 13), and the *block offset* within the region (bits 12 through 7). The lower seven bits are the byte offset within a block and are not used by any RegionTracker structures. We assume only physical addresses are used with RegionTracker.

3.1 Cached Region Hash

The CRH keeps track of those regions that have blocks currently cached. We opt for a simple Bloom-like filter [5] which provides an imprecise representation of the set of regions that are currently cached. It consists of a table of counts which are incremented on each block allocation, and decremented on each eviction. The CRH is indexed using the region number of the block being allocated or evicted. In this work, the index is simply computed as a sufficient number of bits starting from the least significant bit in the region number (e.g., bits 13 through 22 for a 1K entry CRH); however, other indexing functions could be used. This simple, imprecise implementation allows us to use a very small, and hence energy and latency efficient structure to capture *most* first misses. Specifically, the CRH represents a *superset* of all regions that currently have blocks cached. The CRH can also easily be partitioned to further improve energy efficiency.

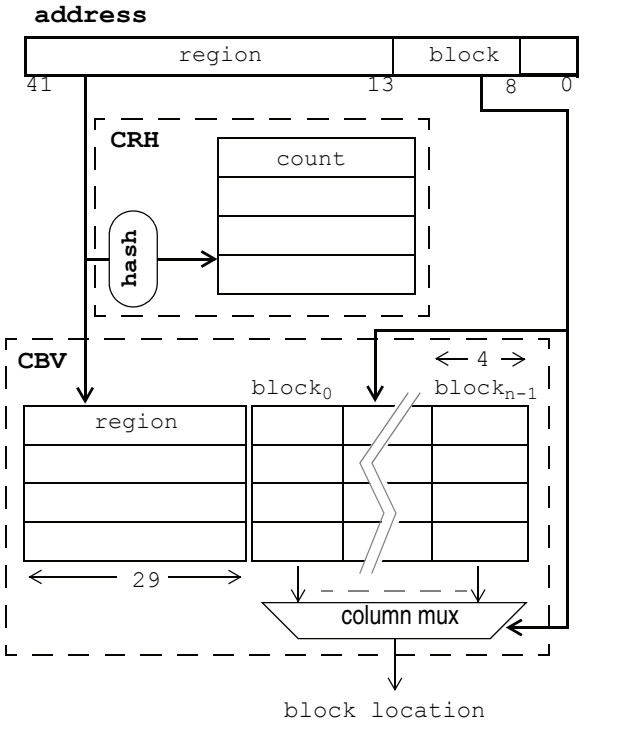

Figure 1: *(a) CRH and CBV organization for 8Kbyte regions, 42-bit physical addresses, 128-byte blocks and an 8-way set-associative cache.(b) An alternative fully-associative CBV implementation that results in lower power and latency.*

When a CRH counter is read, there are two possible outcomes. A counter value of zero indicates a first miss to a region. A non-zero counter value indicates that some portion of that region *may* be cached. The uncertainty results from potential aliasing of different regions onto the same CRH entry. When a first miss is detected, RegionTracker allocates a CBV entry for the region and starts fine-grain tracking of the location of all blocks in that region.

3.2 Cached Block Vector

The CBV is a table where each entry comprises a region tag and a set of information bits for each block within the region. For example, with 8Kbyte regions and 128-byte cache blocks, each CBV entry contains 64 block information fields. In the configurations considered in this work, the information fields encode whether or not the block is cached and where. For an 8-way set-associative cache, four bits are sufficient per block to encode the nine possible states: “not cached” or “cached in way N” where N ranges from 0 to 7. Depending on the cache organization, other information may also be stored in the information fields. For example, the CBV might store status or coherence information, or in a NuRapid memory hierarchy [10], the exact sub-array index can be

stored in the CBV. In the implementations considered in this work, status information is *not* stored in the CBV.

To access the CBV, the region number is compared with the region tags. If a matching entry is found, the information contained in the corresponding block field can be used to access the appropriate data array. The CBV is updated when blocks being tracked are allocated or evicted from the cache so that the CBV block information remains coherent. CBV entries are evicted when space is exhausted and a new entry has to be allocated following the detection of a first miss. Various replacement policies are possible, but this work uses an LRU replacement algorithm. While Figure 1 shows a fully-associative CBV, other organizations are possible as the CBV can be partitioned both vertically and horizontally to reduce energy and latency. Although we do not present the results here, for the programs we studied an 8-way set-associative CBV achieves coverage very close to a fully-associative CBV (within 2%). Further CBV optimizations are possible to reduce energy and latency, but the details are beyond the scope of this work. We note that only four bits need to be read out of the CBV array in Figure 1. Accordingly, only four bitlines are discharged during reads.

3.3 Energy and Storage Requirements

Since each RegionTracker access uses less energy than a conventional tag array access, using RegionTracker can reduce the total lookup energy. Section 5.3 describes the models used to calculate energy used by tag arrays and the RegionTracker structures. The various RegionTracker configurations presented in this paper use between 12% and 16% of the energy of the tag array for each access.

The low energy consumption results from two factors: (1) small size, and (2) small number of bits accessed. The RegionTracker configurations used in this work require between 0.8% and 18% of the storage of the L2 tag array. Appendix A provides a detailed discussion of the storage requirements. In addition to its small size, RegionTracker also benefits from only accessing a few bits on each access. For the configurations studied in this work, each CRH entry is only 10 bits, and only 4 bits in the CBV entry need to be read for each access. This compares to 184 tag bits that need to be read and compared for a 4MB, 8-way set-associative L2 cache with 128-byte blocks and 42-bit addresses.

3.4 RegionTracker Complexity

RegionTracker successfully reduces cache lookup energy with a minimal increase in hardware complexity. Since RegionTracker places no restrictions on how the cache operates, it does not introduce any new complexity in the implementation of the cache itself. Meanwhile, the RegionTracker structures are small and simple, and should lend themselves to a low complexity implementation. Finally, relatively little information needs to be communicated between the cache and RegionTracker; thus, adding RegionTracker to the cache access path should not significantly increase the overall complexity.

4 Related Work

Given the proportion of chip area devoted to caches, many contributions have been made to reducing cache power. However, most existing proposals target level one caches. The filter cache [16], consisting of a small cache placed in front of the L1 cache, can service a large fraction of L1 accesses, but misses to the filter cache incur an increased latency. A similar mechanism has been proposed for increasing L1 bandwidth [35], and [13] explored the idea of using these line-buffers in front of the L2 tag and data arrays to reduce power. Park *et al.,* [26] proposed a simple modification to this scheme which increased its effectiveness. These techniques exploit temporal and fine-grain spatial locality. As shown in Section 5.4, RegionTracker complements these techniques by filtering many L2 accesses that would only be caught by a larger TSB. Specifically, we show that a tiny (two entry) TSB combined with a RegionTracker outperforms a TSB with as many as 128 entries, and we also demonstrate that RegionTracker is more energy efficient.

A number of techniques have also been proposed for reducing the area and power of tag arrays. Decoupled sectored caches [32] and Caching Address Tags [34] are two techniques which reduce the tag array area by sharing tags amongst multiple cache blocks. The resulting structure has fewer tags than cache blocks. This exploits the same spatial locality as RegionTracker. However, since these techniques rely on a reduced number of tags, a single cache miss could require the invalidation of multiple cache blocks because their corresponding tag has been evicted. This incurs not only an initial latency penalty on such a miss, but also a possibly higher overall miss rate which can indirectly impact overall power and performance. RegionTracker does not affect L2 miss rate and, as we report in Section 5.5, with straightforward tuning it never hurts overall performance and hence power. Finally, implementing these techniques requires changing the L2 cache controller, something avoided by the simple RegionTracker implementation.

Other techniques which address tag array power include way prediction [12,14,28] and memoization [20], as well as techniques which attempt to optimize tag search energy using multi-stage tag lookup [8,9]. The former techniques, as well as [4] and [25] apply mostly to the L1 instruction cache, while the latter techniques were demonstrated for the L1 data cache. It is not clear if these techniques will scale well to larger L2 caches with higher

associativities. Additional work has incorporated compiler support for reducing cache power [1,2], and much work has been done which relies on cache partitioning, layout and circuit level techniques to realize energy reduction in caches, including [11, 17, 18, 19, 33].

Bloom filters similar to the CRH have been previously proposed for avoiding snoop-induced tag lookups [23] or snoop broadcasts [24], for L1 hit/miss prediction [27], load/store queue complexity reduction [30] and for miss prediction [22]. Whereas the Bloom filters previously proposed cannot track the locations of individual cache blocks, RegionTracker overcomes this short-coming by combining a bloom-like filter with a fine-grain tracking structure which can track and service most L2 requests.

5 Evaluation

This section is organized as follows: In Section 5.1 we describe our experimental methodology. In Section 5.2 we demonstrate the effectiveness of RegionTracker at servicing lookup requests. In Section 5.3 we report energy savings compared to a standalone, conventional tag array. In Section 5.4, we compare RegionTracker with tag set buffers. Finally, in Section 5.5 we summarize our findings about overall power and performance.

5.1 Methodology

We used Simplescalar v3.0 [7] to simulate the processor detailed in Table 2. Amongst several modifications, we modified the macros for the NOP instruction to not generate memory references (the NOP is a load to register zero and the hardware is supposed to ignore this load) and added support for modelling contention in the memory system. We compiled the SPEC CPU 2000 benchmarks for the Alpha 21264 architecture using HP's compilers and for the Digital Unix V4.0F using the SPEC suggested default flags for peak optimization. All benchmarks were run using a reference input data set. It was not possible to simulate a few benchmarks due to insufficient memory resources. Table 3 presents a list of the benchmarks as well as their memory footprints. Most of these footprints greatly exceed the L2 capacity, thus a reasonable RegionTracker cannot trivially track all blocks for an application.

To obtain reasonable simulation times, samples were taken for one billion committed instructions after skipping the first 100 billion committed instructions. For art and parser we only skipped 20 billion instructions prior to collecting measurements. We experimented with several other one billion instruction samples and with longer samples of up to 40 billion instructions and observed that results did not vary significantly for the different samples. A continuous instruction sample is important for our measurements as RegionTracker structures have to be kept coherent throughout execution. Unless otherwise noted we used timing simulation to measure the overall performance and power impact of RegionTracker. As shown in Table 2, the memory system comprises split level one data and instruction caches, a unified second level cache and a main memory. We studied L2 caches in the range of 2Mbytes to 16Mbytes. In the interest of space and clarity we use an *A/B* naming scheme for RegionTracker configurations where A is the number of CRH entries and B is the number of CBV entries. In all experiments we use an 8Kbyte region size. Section 5.3 describes the details of our power modeling methodology.

Table 2. Base processor configuration

Branch Predictor	**Fetch Unit**
16k GShare +16K bi-modal 16K selector 2 branches per cycle	Up to 6 instr. per cycle 64-entry Fetch Buffer Non-blocking I-Cache
Issue/Decode/Commit	**Scheduler**
any 6 instr./cycle	128-entry/64-entry LSQ
FU Latencies	**Main Memory**
same as MIPS R10000	Infinite, 300 cycles
L1D/L1I Geometry	**UL2 Geometry**
32KBytes, 2-way set-associative with 64-byte blocks	2Mbytes to 16Mbytes, 8-way set-associative with 128-byte blocks
L1D/L1I/L2 Latencies	**Cache Replacement**
3/3/16 cycles	LRU

Table 3. Total simulated memory bytes allocated per application during our simulation interval.

Benchmark	Memory Footprint	Benchmark	Memory Footprint
ammp	27M	***gcc***	133M
applu	186M	***gzip***	185M
apsi	196M	***lucas***	189M
art	89M	***mcf***	186M
bzip2	188M	***mesa***	10M
crafty	2M	***mgrid***	57M
eon	2M	***parser***	62M
equake	50M	***swim***	196M
facerec	17M	***twolf***	3M
fma3d	107M	***vortex***	70M
galgel	45M	***vpr***	51M
gap	193	***wupwise***	181M

5.2 Coverage with Practical RegionTrackers

We first report *coverage* results with RegionTracker. Coverage is the percentage of cache accesses for which RegionTracker provides precise location information, indicating either that the block is not cached, or cached in a specific cache way. We used functional simulation in order to evaluate a wide range of RegionTracker configurations. Figure 2 presents the results of timing simulations for the most promising RegionTracker configurations, showing the average coverage each configuration. Each curve represents a separate configuration, while the x-axis indicates the size of the L2 cache. As expected, coverage increases with larger RegionTrackers and for a fixed RegionTracker

configuration coverage decreases with L2 size. This result implies that on the average RegionTracker is well behaved and can be easily tuned to each cache configuration. Overall coverage varies from as high as 61% to as low as 15%. The results also demonstrate that if the size of the RegionTracker relative to the L2 cache is kept constant, then coverage remains roughly constant as the cache size increases. Consider, for example, a 2K/64 RegionTracker with a 4MB L2 cache, which achieves 45% coverage. Doubling the size of the cache and RegionTracker results in 46% coverage for the 4K/128 RegionTracker with the 8MB cache.

We next demonstrate that although RegionTracker coverage varies significantly across programs, it is high for most. Figure 3 presents the coverage for a 4K/128 RegionTracker for each benchmark with a 4MByte L2 cache. On average, this configuration achieves 55% coverage, with a minimum of 9.5% coverage for vortex and a maximum of 97.6% coverage for gzip. Those programs with low coverage generally access a large number of regions and require larger CBVs to obtain better coverage.

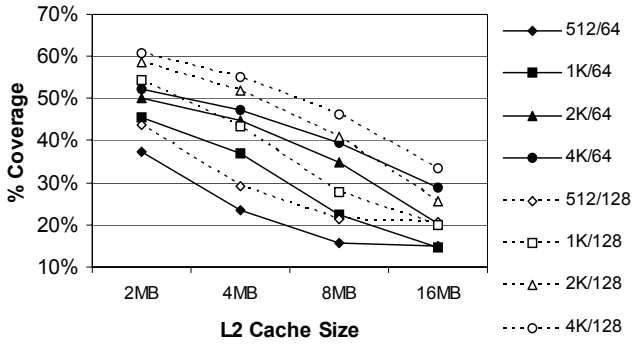

Figure 2: *Average coverage achieved by various RegionTracker configurations (different curves) for various L2 cache sizes (x-axis).*

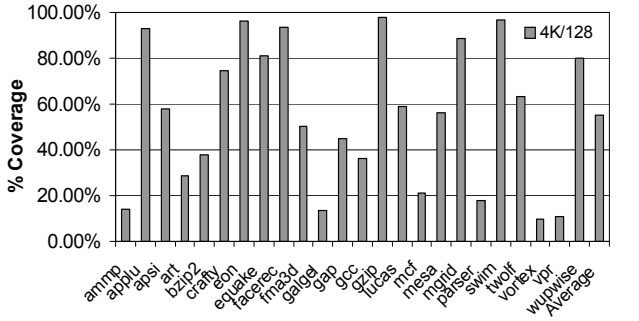

Figure 3: *Per benchmark coverage for a 4K/128 RegionTracker with a 4MB L2 cache.*

5.3 Energy Savings Compared to Conventional Tags

This section demonstrates that significant energy savings are possible with practical RegionTrackers for various L2 caches. We used CACTI 3.2 [29] to model the energy used by the L2 tag arrays and the RegionTracker structures. All structures were modeled in a 65nm technology. We modeled L2 caches from 2MB to 16MB in size. We selected a sub-bank size of 512KB[1], and each cache was divided into the appropriate number of sub-banks. The tag energy was computed as the sum of the tag decode, wordline, bitline, sense amp, and compare energy as reported by CACTI. For the RegionTracker configurations, the CRH was modeled as a direct mapped cache, but the contribution of the tag array energy was ignored as the CRH is an un-tagged structure. The CBV was modeled as a combination of a direct mapped cache and a fully associative CAM structure.

In modelling the various structures, we observed that the sense amps were contributing a significant portion of the energy, especially for the cache tag arrays. Previous work suggests that a power optimized sense amplifier would use about one third of the power used by the sense amplifier modeled by CACTI [21]. We thus scaled sense amp power accordingly. Note that this adjustment reduces the benefits of RegionTracker.

The energy savings were calculated based on the following assumptions:

- It is possible to access a single way in the tag array.
- On an L2 miss, only a single tag way is accessed to write the updated tag information.
- An L2 hit requires a full tag array lookup, and an L2 miss requires (1+1/A) tag array lookups, where A is the L2 associativity (i.e., initial access + single way access to update info).
- Each L2 hit caught by RegionTracker avoids a tag lookup.
- Each L2 miss caught by RegionTracker avoids the initial full tag set access and replaces it with a single tag way access (i.e., 2/A accesses are now required instead of (1+1/A))
- Each L2 access reads both the CRH and CBV.
- Each L2 miss causes a write to the CRH.
- Each L2 miss caught by RegionTracker also writes to the CBV.

These assumptions were used in combination with statistics from the simulations to calculate the lookup energy saved, as a percentage of the lookup energy consumed by a conventional L2 tag array.

Figure 4 reports average energy savings for different RegionTracker configurations (with a CRH with between 512 and 4k entries, and either 64 or 128 CBV entries), represented by different curves, and for L2 cache sizes from 2MB to 16MB, varied along the x-axis. The highest

[1] We evaluated various sub-bank sizes using CACTI and found that 512KB sub-banks minimized the energy-delay product.

savings of 44% is observed for the 4K/128 RegionTracker and the 2Mbyte L2 cache. This RegionTracker produces savings of 38%, 29% and 16% for the 4Mbyte, 8Mbyte and 16Mbyte caches respectively.

Since cache sub-bank size remains constant for all cache sizes, the tag energy remains almost constant; thus, the energy savings are reduced for larger caches according to the reduction in coverage shown in Figure 2. However, as the L2 cache size increases we can also increase the RegionTracker size to maintain the coverage and energy savings. It should be emphasized that as cache capacity increases, larger RegionTrackers become practical as their storage requirements become a smaller fraction of the L2 tag arrays.

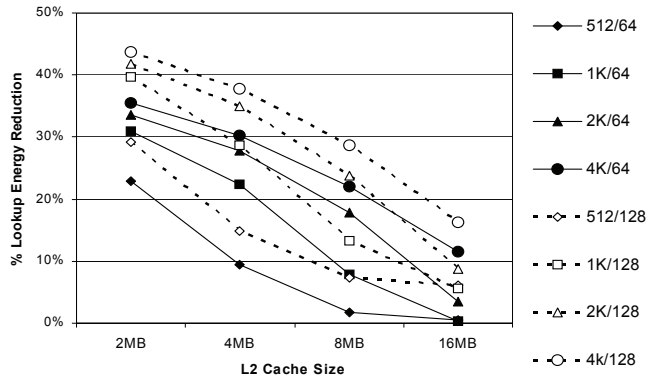

Figure 4: *Average energy savings expressed as fraction over the energy of a conventional L2 tag array. Shown are RegionTrackers with various CRH entry counts and a 64 or 128-entry CBVs. Results are shown for L2 caches of 2Mbytes to up to 16Mbytes (x-axis). Each curve corresponds to a different RegionTracker, labeled CRH/CBV. All RegionTrackers are 8-way set-associative.*

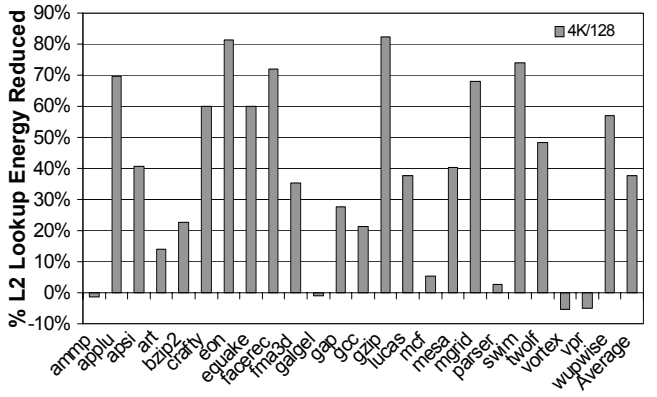

Figure 5: *Per program relative energy savings with a 4K/128 RegionTracker and a 4MB L2 cache, expressed as a fraction over the conventional L2 tag array power.*

Figure 5 indicates that while RegionTracker is robust and provides significant energy reductions for most programs, there are a few programs which exhibit an increase in energy consumption. This figure reports per program energy changes for a 4K/128 RegionTracker with a 4MB L2 cache. For a few programs, RegionTracker increases the lookup energy slightly, with the largest increase of 5% being observed for vortex and vpr. This compares with an average reduction of 38% and a maximum savings of 82% for gzip.

5.4 Comparing with Conventional Tag Set Caching

As we discussed in Section 4, a number of existing proposals for reducing L2 tag power rely either on efficient encoding or on keeping a small cache of recently accessed tags. In this section, we compare RegionTracker with tag set buffers (TSBs), or line buffers as they are often referred to, which have sizes less than or approximately equal to the size of the RegionTracker structure. We compare the two approaches using two metrics, coverage and energy savings. As we explain, TSB coverage rivals that of RegionTracker, however, energy savings does not.

A TSB is a small cache of recently accessed tag sets. For example, for an 8-way set-associative cache, each tag set entry will hold eight tags. Entries are allocated on demand as accesses probe the conventional tag array. Each access first probes the TSB, and if the set it maps to is found in the TSB, then there is no need to access the tags. If the set is not found in the TSB, then it is brought into the TSB after being read from the tag array.

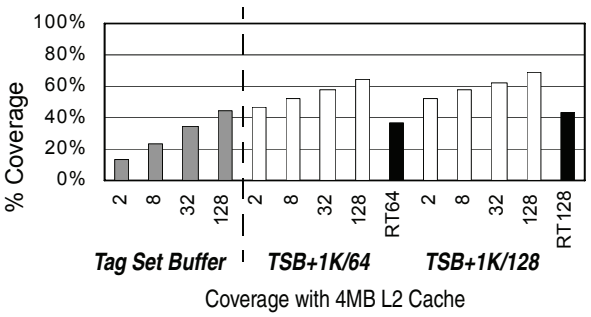

Figure 6: *Comparing Tag Set Buffers of various sizes to RegionTracker in terms of coverage for a 4MB cache. The four grey bars correspond to fully-associative tag buffers of two through 128 sets. The next four white bars are for a 1K/64 RegionTracker combined with Tag Set Buffers with the number of entries reported along the x-axis (two through 128). The coverage of the RegionTracker alone is shown by the next dark bar (RT64). Finally, we double the number of RegionTracker entries to 128.*

Figure 6 reports average coverage for various fully-associative TSBs (range of two to 128 entries), standalone 1K/64 and 1K/128 8-way set-associative RegionTrackers and combinations of the aforementioned TSBs and RegionTrackers. All results in Figure 6 are for a 4MB L2 cache. The grey bars report coverage for TSBs of the corresponding size (listed along the x-axis). The white bars report coverage for hybrid RegionTracker and

TSB organizations. The TSB entry count is listed along the x-axis. The first four are for the 1K/64 RegionTracker and the next four white bars are for the 1K/128 RegionTracker. Finally, the two black bars report coverage with just the 1K/64 (left) and the 1K/128 (right) RegionTrackers. Table 4 reports the storage requirements in bits of the TSBs and the two RegionTrackers as a fraction of the L2 tags. While it appears that a tag buffer can achieve coverage comparable to, or better than, a similarly sized RegionTracker, Section 5.4.1 shows that RegionTracker obtains better energy reduction.

Table 4. Comparing the storage requirements of tag buffers and RegionTrackers. Storage requirements are measured in bits and reported as a fraction over that of a conventional tag array for a 2Mbyte cache.

Tag Set Buffer	*Storage Requirements (bits)*
2	< 1%
4	< 1%
8	< 1%
16	< 1%
32	1.6%
64	3.3%
128	6.5%
CBV 64 + 1K CRH	6.6%
CBV 128 + 1K CRH	10.9%

5.4.1 Tag Set Buffer vs. RegionTracker trade-offs

A number of factors suggest that RegionTracker might offer a number of advantages over TSB. The designs of RegionTracker and TSBs are drasctically different. We used CACTI to model various TSBs, and while TSB may be conceptually simpler, each TSB access utilizes more energy than each RegionTracker access. Thus, a TSB with a given coverage will likely use more energy than a RegionTracker configuration that acheives similar coverage.

Figure 7 shows the average energy reduction for tag set buffers with 16, 32, 64 and 128 sets (different curves) for L2 caches of various sizes (x-axis). Only two configurations actually reduce energy on average, and the maximum reduction is only 3% for a 128 set TSB with a 16MB cache. Contrary to RegionTracker, TSB energy savings generally increase with cache size as the number of tag bits decreases for larger caches, and TSB coverage remains roughly constant as the cache size increases.

RegionTracker has an additional set of potential advantages over tag set buffers. These include the increased flexibility of RegionTracker implementations. Most accesses to RegionTracker involve a very small number of bits compared to the 180 or so tag bits involved in a TSB access. This leads to a flexibility in how the RegionTracker structures can be implemented, and where they are located. Also, RegionTracker would be relatively easy to port to novel or unconventional cache architectures such as NuRapid [10] or skewed

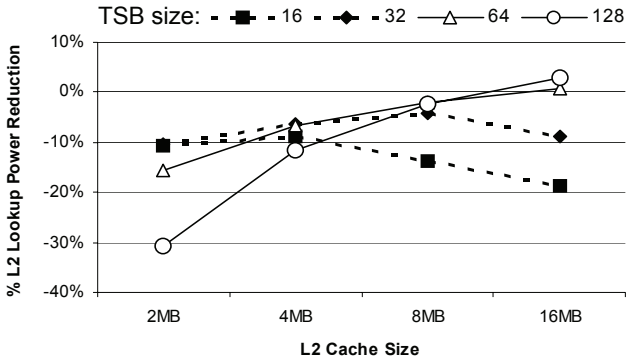

Figure 7: *L2 lookup power reduction (increase) with tag buffers of various sizes.*

associative caches [31]. An investigation of these issues is beyond the scope of this paper. The few results presented indicate that while both TSBs and RegionTracker can achieve comparable coverage, RegionTracker provides a much larger energy reduction than TSBs.

5.5 Performance and Overall Power

As mentioned above, RegionTracker affects L2 latency, and thus impacts both overall performance and power. We have measured the overall performance impact of RegionTracker configurations with 64 or 128 CBV entries and 512, 1K, 2K, and 4K CRH entries, assuming that it decreases L2 access latency by two cycles on a RegionTracker hit while it increases it by one cycle for a RegionTracker miss. These assumption were validated using an analytical latency model based on CACTI [29]. We studied caches of 2Mbytes and 4Mbytes. On average, overall performance increased less than 1% with RegionTracker. In the best case of twolf, performance increased by 2% with a 128/4k RegionTracker and a 2MB cache. Only a few benchmarks suffered from decreased performance, with the worst case being fma3d which had a slowdown of 0.02% with a 128/512 RegionTracker and a 4MB L2 cache. Correspondingly, overall processor power decreased slightly on average with the RegionTracker configurations we studied, although a few benchmarks saw increases of less than 0.1% for some configurations.

6 Summary

We proposed RegionTracker as an area, power and latency efficient implementation of memory hierarchy lookup structures aimed primarily at higher-level, relatively large, on-chip caches.

RegionTracker implements the concept of dual-grain tracking, using a simple Bloom-like filter (CRH) to track coarse-grain regions, combined with a small table of fine-grained region tracking entries (CBV). A key result was the demonstration that using a dual-grain tracking

approach provides significantly more potential than simple, demand-based allocation of fine-grained tracking resources. We demonstrated the utility of RegionTracker for reducing power and latency for L2 tag lookups. A 2k/128 RegionTracker saves 35% of the tag lookup power for a 4Mbyte L2 cache, while requiring less than 7% of the resources required for the conventional tag array. RegionTracker can be complemented by adding a tiny tag set buffer to achieve better coverage than either RegionTracker or tag set buffers can provide on their own. Other potential applications of RegionTracker include increased tag lookup bandwidth for aggressive prefetching, or increasing L1 tag port bandwidth and lookup latency, although the latter application would involve complex scheduling and latency issues.

Acknowledgements

The authors would like to thank Patrick Akl, Ioana Burcea, Davor Capalija, Elham Safi, and the anonymous reviewers for their valuable comments. Jason Zebchuk is supported in part by an NSERC Canada Graduate Scholarship (CGS-M). This work was supported by Semiconductor Research Corporation under contract #901.001, an NSERC Discovery Grant, a Canada Foundation for Innovation equipment grant, an Intel equipment donation and an Intel research grant.

7 References

[1] D. H. Albonesi. *Selective cache ways.* In the Proc. of the 32nd Annual International Symposium on Microarchitecture, Nov. 1999.

[2] R. Ashok, S. Chheda, C. A. Moritz, *Cool-Mem: Combining Statically Speculative Memory Accessing with Selective Address Translation for Energy Efficiency,* In the Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2002

[3] B. Bateman, C. Freeman, J. Halbert, K. Hose, and E. Reese. *A 450Mhz 512KB Second-Level Cache with a 3.6GB/S Data Bandwidth.* In the Proc. of the IEEE International Solid-State Circuits Conference, 1998.

[4] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. A*rchitectural and compiler support for energy reduction in the memory hierarchy of high performance processors.* In the Proc. of the International Symposium on Low Power Electronics and Design, Aug. 1998.

[5] B. Bloom. *Space/time trade-offs in hash coding with allowable errors.* Communications of ACM, pages 13(7):422-426, July 1970.

[6] W.J. Bowhill et al. *Circuit Implementation of a 300Mhz 64-bit Second Generation Alpha CPU.* Digital Journal vol. 7., 1995.

[7] D. Burger and T. Austin. The Simplescalar Tool Set v2.0, *Technical Report UW-CS-97-1342. Computer Sciences Department, University of Wisconsin-Madison*, June 1997.

[8] D. Brooks, V. Tiwari M. Martonosi. *Wattch: A Framework for Architectural-Level Power Analysis and Optimization*. In the Proc. of the 27th Annual International Symposium on Computer Architecture, June 2000.

[9] Y.-J. Chang, S.-J. Ruan and F. Lai, *Design and analysis of low-power cache using two-level filter scheme,* IEEE Transactions on VLSI, vol 11, no. 4, Aug. 2003.

[10] Z. Chishti, M. D. Powell and T. N. Vijaykumar, *Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures,* In the Proc. of the 36th Annual International Symposium on Microarchitecture, Dec. 2003.

[11] M. Huang, J. Renau, S.-M. Yoo and J. Torellas. *L1 Data Cache Decomposition for Energy Efficiency*. In the Proc. of the International Symposium on Low-Power Electronics and Design, Aug. 2001.

[12] K. Inoue, T. Ishihare and K. Murakami. *Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption,* In the Proc. of the International Symposium on Low-Power Electronic Design, August 1999.

[13] M.B. Kamble and K. Ghose. *Reducing Power in Superscalar Processors using subbanking, multiple line buffers and bit-line segmentation.* In the Proceedings of the International Symposium on Low Power Electronics and Design, 1999.

[14] R. E. Kessler, R. Joss, A. Lebeck, and M. D. Hill, *Inexpensive Implementations of Set-Associativity,* In the Proc. 16th Annual International Symposium on Computer Architecture, June 1989.

[15] C. Kim, D. Burger and S. W. Keckler, An Adaptive, *Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches,* In the Proc. of the 10. International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2002.

[16] J. Kin, M. Gupta, and W. Mangione-Smith. *The Filter Cache: An Energy Efficient Memory Structure*. In the Proc. of the 30th International Symposium on Microarchitecture, pages 184-193, Nov. 1997.

[17] U. Ko, P. T. Balsara, and A. K. Nanda. *Energy Optimization of Multi-Level Processor Cache Architectures.* In the Proc. of the International Symposium on Lower Power Design, Aug. 1995.

[18] U. Ko, P. T. Balsara, and A. K. Nanda. *Energy Optimization of Multilevel Cache Architectures for RISC and CISC Processors.* In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6(2), Jun. 1998.

[19] H.S. Lee and G. S. Tyson. *Region-Based Caching: an energy-delay efficient memory architecture for embedded processors.* In the Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Nov. 2000.

[20] A. Ma, M. Zhang, and K. Asanovic, *Way Memoization to Reduce Fetch Energy in Instruction Cache,* Workshop on Complexity-Effective Design, held in conjunction with the 28th Annual International Symposium on Computer Architecture, June 2001.

[21] M. Margala. *Low-power SRAM circuit design*. Memory Technology, Design and Testing, 1999. Records of the 1999 IEEE International Workshop on 9-10 Aug. 1999.

[22] G. Memik, G. Reinman, W. H. Mangione-Smith, *Just Say No: Benefits of Early Cache Miss Determination,* In the Proc. of the 9th International Symposium on High-Performance Computer Architecture, Feb. 2003.

[23] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. *JETTY: Filtering snoops for reduced energy consumption in SMP servers.* In the Proc. of the 7th International Symposium on High- Performance Computer Architecture, January 2001.

[24] A. Moshovos, *RegionScout: Exploiting Coarse-Grain Sharing in Snoop Coherence,* In the Proc. 32nd Annual International Symposium on Computer Architecture, June 2005.

[25] R. Panwar and D. Rennels. *Reducing the frequency of tag compares for low power I-Cache design.* In the Proceedings of the International Symposium on Low Power Electronics and Design, Aug. 1995.

[26] W.-H. Park, A. Moshovos, and B. Falsafi. *ReCast: Boosting Tag Line Buffer Coverage in Low-Power High-Level Caches 'for Free',* In the Proc. of the 2005 IEEE International Conference on Computer Design, Oct. 2005.27, Nov. 2000.

[27] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark and K. Lai, *Bloom filtering cache misses for accurate data speculation and prefetching,* In the Proc. of the 16th International Conference on Supercomputing, June 2002.

[28] 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 the Proc. of the 34th Annual Symposium on Microarchitecture, Dec. 2001.

[29] G. Reinman and N.P. Jouppi. *An Integrated Cache Timing and Power Model. Technical report,* COMPAQ Western Research Lab, 1999.

[30] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore and S. W. Keckler, *Scalable Hardware Memory Disambiguation for High-ILP Processors,* In the Proc. 36th Annual International Symposium on Microarchitecture, Nov. 2003.

[31] A. Seznec. *A Case for Two-Way Skewed-associative Caches.* In the Proc. of the 20th Annual International Symposium on Computer Architecture, May 1993.

[32] A. Seznec. *Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio*. In the Proc. of the 21st Annual International Symposium on Computer Architecture, June 1994.

[33] C. Su and A. Despain. *Cache Designs for Energy Efficiency.* In the Proceedings of the 28th Annual Hawaii International Conference on System Sciences, pages 306-315, 1995.

[34] H. Wang, T. Sun, and Q. Yang. *CAT – caching address tags: A technique for reducing area cost of on-chip caches.* In the Proc. of the 22nd Annual International Symposium on Computer Architecture, June 1995.

[35] K. M. Wilson, K. Olukotun, and M. Rosenblum. *Increasing cache port efficiency for dynamic superscalar microprocessors*. In the Proc. of the 23rd Annual International Symposium on Computer Architecture, May 1996.

Appendix A RegionTracker Relative Storage Requirements

Since RegionTracker acts to supplement a conventional tag array to reduce energy, it should require minimal overhead in terms of on-chip area. Tables 5 and 6 report the storage requirements (total bit count) of various CRH and CBV structures respectively, demonstrating that reasonably sized RegionTracker structures are much smaller than conventional tag arrays. The storage requirement of each structure is expressed as a fraction of the storage requirement of the tag array of a 2Mbyte L2 cache. This provides a first-order approximation of the area cost. The overall bit requirement is meaningful as an area estimate as it remains constant regardless of implementation details, such as partitioning into separate banks or sub-arrays.

Table 5 shows CRH requirements for entry counts of 512 through 4K. The size of each CRH entry depends on the cache configuration and region size. In general, each block in the cache could map to the same CRH entry as a result of aliasing. However, with the simple indexing function used in this work, only a fixed number of cache sets will can map to any CRH entry. Thus the number of bits required for each entry is only *N x lg(L2 Associativity x (Region size / block size))*, where *N* is the number of CRH entries.

Table 5. CRH storage requirements as a fraction of the bits required by the tag array of a 2Mbyte 8-way set-associative L2 cache with 128-byte blocks

CRH entries	*Storage*
512	1.2%
1K	2.4%
2K	4.8%
4K	9.6%

Table 6. Eight-way set-associative CBV storage requirements as a fraction of the bits required by the tag array of a 2Mbyte, 8-way set-associative L2 cache with 128-byte blocks. Ratios are shown for different CBV entry counts and region sizes. We assume 42-bit physical addresses and two status bits per tag entry (fractions will improve if additional status bits were used).

CBV Entries	*Region Size in Bytes*						
	512	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**
16	<1%	<1%	<1%	<1%	**1.1%**	2.0%	3.9%
32	<1%	<1%	<1%	1.2%	**2.1%**	4.0%	7.9%
64	<1%	1.0%	1.4%	2.3%	**4.3%**	8.1%	15.8%
128	1.4%	1.8%	2.7%	4.6%	**8.4%**	16.1%	31.5%
256	2.6%	3.5%	5.4%	9.2%	**16.8%**	32.2%	62.8%
512	5.2%	7.0%	10.7%	18.3%	**33.5%**	64.2%	125.6%

The CBV storage requirements are primarily proportional to the number of CBV entries, the number of blocks within the region and the number of L2 ways. Larger regions or smaller blocks results in more block information fields in each CBV, and the L2 associativity determines the size of each field. The size of the region tags has only a small effect on the total CBV size. As shown in Table 6, a 128-entry CBV with 8Kbyte regions requires less than 9% of the bits needed by the conventional tag array. The percentages shown in Table 6 can also be used to estimate the relative cost of RegionTracker for larger caches since CBV requirements are not directly affected by cache size. For example, as the cache size doubles, the tag array approximately doubles in size as well, thus halving the relative storage requirements of the CBV. This work considers caches in the range of 2MB to 16MB, so while a 512 entry CBV requires 125% of the storage of a 2MB L2 tag array, it requires only 4.76% of the bits required by a conventional tag array for a 16MB cache.