RegionTracker: A Case for Dual-Grain Tracking in the Memory System

Jason Zebchuk and Andreas Moshovos
Computer Group Technical Report
February 23, 2006
Electrical and Computer Engineering
University of Toronto
{zebchuk, moshovos}@eecg.toronto.edu

Abstract

This work proposes area, power and latency efficient implementations of memory hierarchy lookup structures aimed primarily at higher-level, relatively large on-chip caches. The mechanisms proposed provide location information for a large fraction of cache references eliminating the corresponding accesses to a much larger, slower and power demanding conventional tag array. The key contribution of this work is 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. At the first level, a coarse-grain structure tracks which large regions of memory have blocks currently cached and uses this information to detect newly accessed regions. A second-level fine-grain mechanism tracks the location of all the blocks within a few newly accessed regions as these are identified by the first level. We demonstrate that our dual-grain tracking significantly outperforms the conventional, demand-based allocation of tracking resources and propose RegionTracker, *an implementation of dual-grain tracking. RegionTracker is simple, can be easily partitioned for optimizing its power and latency, does not use cascaded lookups and does not impose any restrictions on cache placement. RegionTracker can capture a large fraction of memory references for various unified L2 caches. For example, we show that a RegionTracker that uses just 5% of the storage used by a conventional tag array and that can track just 128 8Kbyte regions, is able to capture 56% of all memory references in a 4Mbyte L2 cache and reduce L2 lookup power by 38% on the average. We also demonstrate that RegionTracker can outperform and complement conventional, demand-driven tag set buffers.*

1 Introduction

Multiple semiconductor technology and application trends support two observations which motivate this work: (1) the number of accesses seen by higher-level (level two or higher) caches will increase, and (2) the size of these caches will also increase. These two observations suggest that the power and latency of higher-level caches will increase both in absolute and relative terms and so will the demand for lookup bandwidth. To mitigate the effects of the aforementioned observations this work proposes area, power and latency efficient implementations of memory hierarchy lookup structures aimed primarily at high-level, relatively large, on-chip caches.

Historically, application memory footprints and working sets have been increasing as “typical” applications evolve (i.e., new applications are introduced or the functionality of existing applications is enhanced). Moreover, processing speeds have also being increasing. Main memory speeds, however, have been increasing at a much lower rate resulting in an increasing speed gap between on-chip processing rates and main memory latencies. Using larger level one caches would be a straightforward remedy. Unfortunately, increasing level one cache size beyond a certain point is typically not possible without hurting performance; while semiconductor technology improvements facilitate faster transistors, the amount of SRAM storage that can be accessed within a reasonable number of cycles (e.g., one to three) remains limited because processing speeds also increase. It is for these reasons that additional, larger caches have been introduced over the years, and there is strong evidence to suggest that the trend towards larger higher-level caches will continue. In addition to increased program references two other factors contribute to increased demand for on-chip, higher-level cache lookup bandwidth: First, as main memory speeds lag behind, prefetching becomes increasingly important for maintaining performance improvements. Prefetching produces additional references, thus increasing demand for on-chip lookup bandwidth. Second, as more processing cores are incorporated onto the same die, coherence related traffic increases, creating even more demand for cache lookup bandwidth more so for snoop-coherence-based implementations.

To offset the effects of these trends, we seek to complement conventional tag arrays with small, faster, more power-efficient structures which can provide location information for a large fraction of cache references. The key contribution of this work is the concept of *dual-grain tracking,* or DGT for short. In DGT two levels inter-operate to achieve high coverage. A first, *coarse-grain* tracking level identifies newly touched *regions* of memory (where a region is a continuous, aligned area of memory whose size is a power of two). Once a new region is touched by the program, the second-level takes over and tracks the location of individual cache blocks within the region. An important result of this work is that using the first-level, coarse-grain tracking performs significantly better than a conventional, demand-driven policy. We also propose *RegionTracker,* a practical implementation of DGT. The first, coarse-grain level in RegionTracker uses imprecise information to capture most newly touched regions. This use of imprecise information leads to a small, fast and power efficient implementation. The second level uses a simple, small table design to track fine-grain information for a few regions. Both levels can be easily partitioned to further improve speed and power. They impose no additional restrictions on data placement and are relatively fast as they do not use cascaded lookups.

This work makes the following contributions: (1) It demonstrates that DGT has great potential in servicing many of the accesses to higher-level caches, even when these caches become relatively large (e.g., 16Mbytes). (2) It proposes *RegionTracker*, an area, power and latency efficient implementation of DGT and demonstrates that practically sized RegionTrackers can capture many requests and reduce power significantly (e.g., 45% of lookup power for a 2Mbyte cache with a RegionTracker that requires just 12.4% of the resources required by a conventional tag array). (3) It demonstrates that dual-grain tracking significantly outperforms a naive, demand-driven allocation of fine-grain lookup resources. (4) Finally, it shows that RegionTrackers are preferable to tag set buffers, more so if a very small tag set buffer (e.g., two entries) is combined with them.

The rest of this paper is organized as follows: In Section 2 we briefly discuss DGT and the intuition behind it, and then we present experimental evidence in support of two observations: (1) there is significant potential for DGT methods, and (2) DGT is superior to naive, demand-based allocation of lookup resources. In Section 3 we present RegionTracker, an implementation of DGT, and discuss its application for level-two, or L2 tag lookup power reduction. 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 bandwidth amplification and power 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 Concept and Motivation

If a structure much smaller than a conventional tag structure can provide the same information for a sufficient fraction of memory access, then it would provide increased area, power and latency efficiency. For example, tag line buffers, e.g. [13], are a small cache of recently accessed tags or tag sets which rely on temporal and spatial locality in the post-L1 memory stream to service many memory requests. Their utility for higher level caches is gradually reduced as lower level caches capture much of this locality.

The structures proposed in this work achieve high 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 but only for a few, large memory regions. Specifically, the first, coarse-grain, level aims at detecting *first misses* at the region level. 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 second level, fine-grain tracking mechanism is used to track the location of all blocks within the region. This can be done, for example, by recording whether or not a block is cached, and if so, in which data way it is cached. It is important to observe that when a first miss is detected, *complete* location information is also detected for the entire region, since none of the blocks within the region are currently cached. In this way, a single access uncovers information for many blocks, making DGT effective despite a lack of temporal locality. As an extreme example, when all accesses are misses, conventional, demand-driven methods such as tag set buffers would perform poorly. Dual-grain tracking, on the other hand, relies on spatial locality, and thus could capture *all* of these accesses (the first miss per region can be captured by the first level and all subsequent accesses by the second-level tracking mechanism — in Section 3.3 we explain how power and latency can be reduced even for L2 misses).

DGT was designed primarily to exploit a behavior that is typical of many applications with large memory footprints. Specifically, while these applications access a very large set of regions over their lifetime, many of them only operate on a few large memory regions at any given time. The first time an application accesses a region, it incurs a *first miss*, giving DGT an opportunity to track subsequent references within that region. Assuming that only a few regions are accessed at a time, DGT should be able to track them all successfully. 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 DGT again has the opportunity to detect it and start tracking the region.

The aforementioned behavior is typical of numerical applications that access several large arrays sequentially and repeatedly. By tracking a few regions at the time, DGT will be able to track the location of the blocks belonging to different arrays. The next time the same arrays are processed, all previously accessed blocks will have been evicted, assuming that the arrays are much larger than the cache. Accordingly, new first misses can be detected so those regions can again be tracked. While reasoning about numerical applications with regular access patterns is relatively easy, non-numerical applications, with less regular access patterns, often exhibit similar behavior. Due to temporal and spatial locality, a large fraction of their references fall under only a few regions at any one time, and hence they can be tracked by DGT. Assuming a large memory footprint, by the time the same data or instructions are accessed again it is very likely that the corresponding blocks will have been evicted due to capacity misses. There are many other scenarios under which DGT offers significant benefits, yet the previous discussion illustrates the key intuition behind this approach.

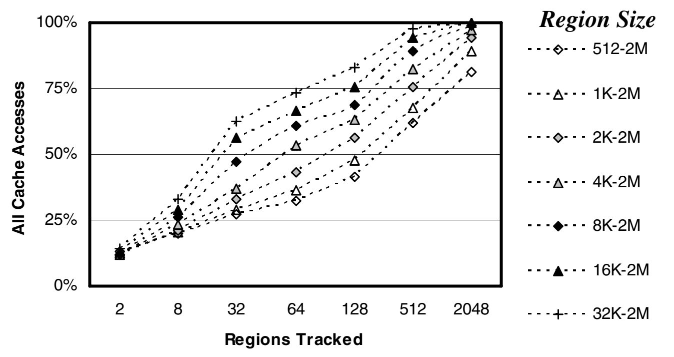
2.1 Potential

Before presenting the RegionTracker design, we provide experimental evidence of the potential of DGT methods. Specifically, the results of this section support the following two observations: (1) there is significant potential for DGT assuming perfect first miss detection, and (2) first miss detection is crucial in exposing this potential, as a simple, demand-driven allocation of fine-grain tracking resources exhibits significantly lower potential. In support of the first observation, we report the average *first-miss-region-locality,* or *FMRL* for typical programs. We define *FMRL(N)* as the number of cache accesses that could be serviced (i.e., we know where the block is) if we were able: (1) to perfectly detect first misses for a given region size, and (2) to fine-grain track the last *N* unique regions that had a first miss detected. The higher the FMRL(N) for small values of N, the higher the potential of DGT. In support of the second observation, we compare FMRL(N) with *region-locality(N),* or *RL(N)*, which is the fraction of cache accesses that could be serviced if we track the last *N* unique regions that were accessed. This corresponds to the potential of conventional, demand-driven allocation of lookup resources.

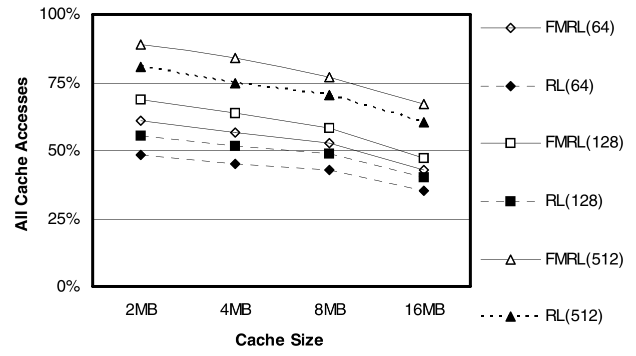
There are many parameters that influence FMRL(N) including program behavior, cache geometry, cache size, and region size. For the purpose of this discussion we focus on cache size, ignoring associativity (our experiments have shown that FMRL is not noticeably sensitive to changes in the associativity), we consider regions in the range of 512 bytes and up to 32 Kbytes and vary N, the number of regions that are fine-tracked, from two to up to 2K. Presenting results for all combinations of cache size, region size and the number of fine-tracked regions is impractical. Instead, we present a subset of the results to illustrate a few key trade-offs. First, we fix the cache size and study how FMRL varies as a function of region size and the number of fine-tracked regions; then, we fix the region size and vary the cache size.

Figure 1(a) shows the average FMRL(N) over all programs studied and for various values of N (x-axis) (i.e., fine-grain tracked regions), and region sizes (separate curves) for an 2 Mbyte L2 cache. Our experimental methodology is discussed in Section 5.1. The number of cache accesses that can be potentially serviced increases with the region size and the number of fine-grain tracked regions albeit the increase becomes more pronounced as the number of fine-tracked regions increases. Both parameters effectively increase the reach of the second-level fine-grain tracking mechanism. Increasing the region size increases the area of memory that can be tracked even if the number of regions tracked remains constant. These measurements show that significant potential exists for DGT. For example, by just tracking 32 8Kbyte regions it is possible to capture approximately half of all memory references (46%), whereas tracking 64 32Kbyte regions captures about three out of four references (74%).

Figure 1(b) reports FMRL(N) for three practical values of N (64, 128 and 512) and for caches of various sizes (2Mbytes up to 16Mbytes). We use a region size of 8K (common OS page size). The three values of N were selected as they correspond to RegionTracker implementations whose storage requirements are a small fraction of the tags for the largest cache used in this experiment (see Tables 2 and 3 in Section 3). The potential for DGT drops for larger caches, when the number of fine-grain regions tracked remains constant (i.e., DGT resources remain constant). But even with just 64 fine-grain tracked regions there is potential to capture 45% of all references to a 16M cache. When the amount of RegionTracker resources remains a constant fraction of the L2 tags the potential does not drop with cache size and in some caches increases. This can be seen by comparing for example the potential with 64 fine-grain tracked regions for the 2M cache (61%) to that with 512 fine-grain tracked regions for the 16M cache (67%).



(a) *DGT potential for a 2M L2 cache*



(b) *DGT and demand-driven potential for various cache sizes*

Figure 1: *DGT potential: Shown is the first miss region locality, FRML(N) (see text for definition) as: (a) a function of region size (different curves) and of the regions that we can fine-track (x-axis) and for a 2 Mbyte, 8-way set-associative L2 cache, and (b) a function of cache size (x-axis) for an 8k region and for 64, 128 or 512 fine-tracked regions (curves). Part (b) also shows the potential of a demand-driven tracking method for the same number of fine-tracked regions (region-locality(N) or RL(N) — see text for the definition).*

Finally, we present experimental evidence of an observation that is central to this work: DGT can greatly boost potential when there is a limited amount of fine-grain tracking resources. Figure 1(b) compares the potential of naive, demand-driven tracking (expressed as RL(N)) with that of DGT (expressed as FMRL(N)) for values of N of 64, 128 and 512, and for 2Mbyte to 16Mbyte caches. These measurements show that the potential with DGT is always significantly higher, especially for smaller values of N (i.e., when there are fewer fine-grain tracking resources). For example: (1) the potential with DGT with just 64 regions is consistently higher than that with demand-driven tracking with 128 regions, and (2) for a 2Mbyte cache FMRL(64) is 61% which is 13% higher than RL(64) and 6% higher than RL(128). The demand-driven method considered in this section uses the same fine-grain region-level tracking resources as DGT. In Section 5.4 we compare RegionTracker with a conventional tag set buffer demonstrating the benefits of DGT compared to a demand-driven method that uses set-grain tracking resources.

In summary, this section has demonstrated that dual-grain tracking has significant potential which does not diminish as the on-chip cache sizes increase, and that its potential is significantly higher than that of naive, demand-based allocation of tracking resources.

3 RegionTracker Design and Application

RegionTracker is an implementation of DGT. Conceptually, DGT comprises two structures, one that tracks caching information at the coarse-grain level of a region and one at the fine-grain level of cache blocks per region. The RegionTracker implementations studied in this work mirror this organization and 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 [23]. The CBV is used to track 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 2(a). Both structures are indexed using parts of the incoming address. Without loss of generality, in the discussion of 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 physical addresses.

3.1 Cached Region Hash

The CRH keeps track of those regions that have blocks currently cached. A first miss is detected when an access determines that there are currently no other cached blocks in the same region. We opt for a simple Bloom-like filter [5] which provides an imprecise representation of the set of regions that are currently cached. This allows us to use a very small, and hence power and latency efficient, structure to capture *most* first misses. Specifically, the CRH represents a *superset* of all regions that currently have blocks cached. It provides two responses: (1) the region is definitely not cached, and (2) the region *may be* 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 part of the address 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 part of the address (e.g., bits 13 through 22 for a 1K entry CRH); however, other indexing functions could be used. To determine the first miss, the CRH is accessed and the corresponding count is read. If the value is zero then there are no cached blocks from the same region. If the value is non-zero, then there may be cached blocks. The uncertainty is due to potential aliasing of different regions onto the same CRH entry. Once 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. Comparing the results of Sections 5.2 and 2 demonstrates that small CRHs (512 to 1K entries) offer most of the benefits of ideal first miss detection. The CRH can be partitioned to improve power efficiency.

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 can be “0” through “7”. Depending on the cache organization, other information may also be stored. For example, status information such as whether the block is dirty or coherence information can be stored in each block information entry, or, in a NuRapid memory hierarchy [10], the exact sub-array index can be stored into the CBV. In the implementations considered in this work status information is not stored in the CBV (see Section 3.3).

To access the CBV, the region portion of the address 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 (we assume serialized cache tag and data accesses for power efficiency [6]). The CBV is updated when blocks 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. While Figure 2(a) shows a fully-associative CBV other organizations are possible as the CBV can be partitioned both vertically and horizontally to improve power and latency. As reported in Section 5.2 for the programs we studied an 8-way set-associative CBV performs very close (within 2%) of a fully-associative CBV.

The power and latency of the CBV organization shown in Figure 2(a) can be improved. Instead of using a multiplexer to select the appropriate block information after a full CBV entry has been read out, as shown in part (a), we can instead use column select signals to activate only the appropriate column (the block offset is known in advance). Latency can be further improved with the organization shown in part (b). Here the region tags and the block information bits have been separated into two structures. The block information bits are rotated so that each CBV entry forms a single column. This way each row contains the block information for all blocks at a specific offset within the regions. The advantage is that the region tag matching and the per block information access can proceed in parallel. At the end, if a region tag match is found a multiplexer selects the appropriate CBV entry. For this multiplexer we can use the column multiplexers of the SRAM array further reducing power. At the end just four sense amplifiers are needed. This is a lot less than those needed by the regular tag array (we need to read eight tags plus the status bits and then compare). In Section 5.3 we show that 8-way set-associative CBVs result in significant power savings compared to conventional tags. Each partition of these CBVs is organized as shown in Figure 2(b) with eight region tags and the appropriate block information entries (e.g, 64 for an 8Kbyte region and 128-byte blocks).

3.3 An Application: Reducing Power and Latency for L2 Lookups

RegionTracker has several potential applications. For example, it can be used to amplify tag lookup bandwidth in support of aggressive prefetching or to compensate for coherence related lookups in a multi-core architecture. In this work we demonstrate its utility for reducing power and latency for L2 tag lookups. In this application a RegionTracker is placed in front of a conventional L2 tag array. As accesses are issued by the L1, they first inspect the RegionTracker, accessing both the CBV and the CRH in parallel. If sufficient information is found in the RegionTracker, then we may be able to completely avoid an L2 tag lookup, or reduce the amount of L2 tag resources that have to be probed. At the end, the L2 data arrays are accessed to complete the access. We assume an in-series tag and data array organization for the L2 to reduce power, as several commercial designs have done, e.g., [6].

Table 1 summarizes how RegionTracker impacts L2 tag power for all possible scenarios; and, it also lists what parts of the L2 tag array have to be accessed. Power and latency can be reduced on a RegionTracker *hit* which

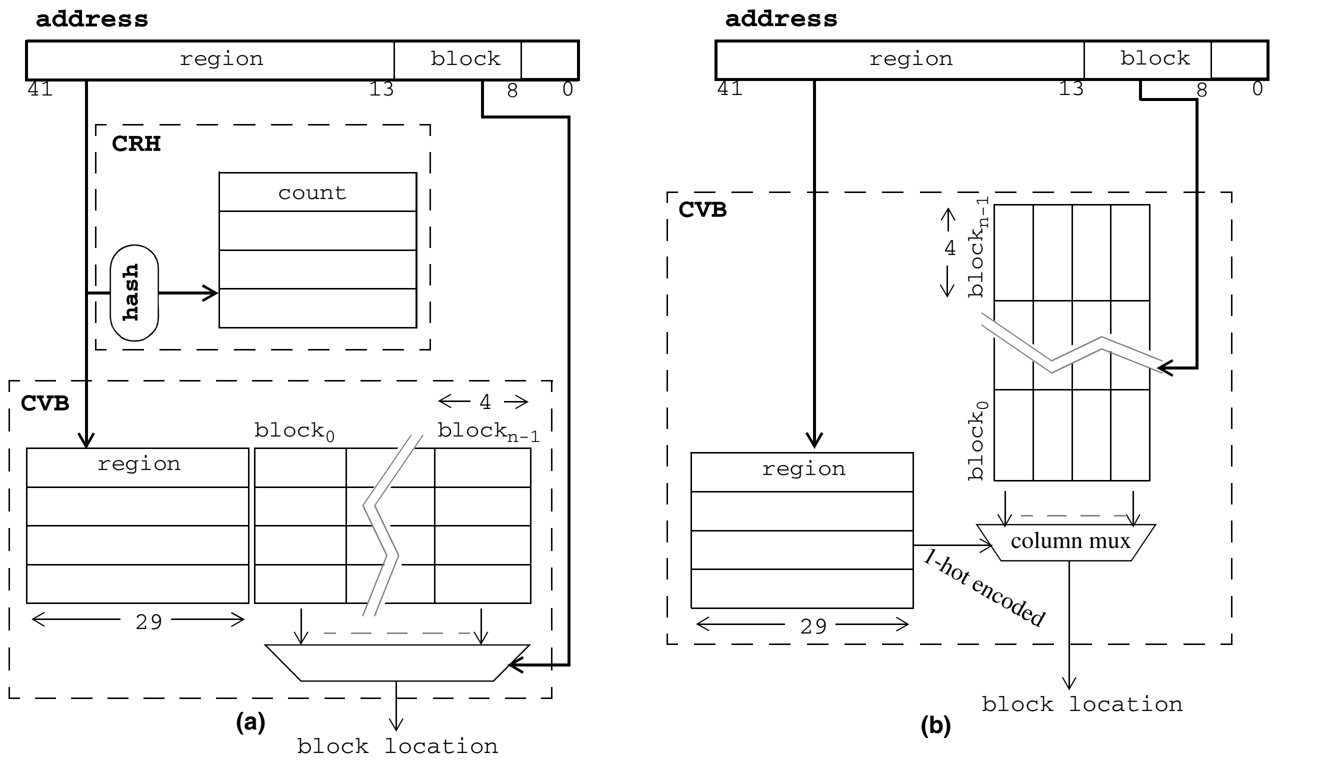


Figure 2: *(a) CRH and CBV organization for 8Kbyte regions, 42-bit physical addresses, 128-byte blocks and an 8-way set-associative cache.(b) An alternative fully-associative CBV implementation that results in lower power and latency.*

occurs not only when the CBV contains information about the specific block, but also when the CRH detects a first miss to a region. In the first case, the CBV entry contains the actual data way where the corresponding block resides, so the L2 tag arrays only need to be accessed to update status information as needed (e.g., on a read there is no need to update the L2 stored status — however, we do have to update the relevant replacement information). Since we know which way the block resides in, we can update just the status bits for the particular tag using column select lines in the tag array, or by assuming that status information is stored separately. If no information is found in the CBV but the CRH detects this access as a first miss, then we know that the block is not cached. Therefore, we only need to access the replacement information for the specific tag set and update a single tag (as opposed to accessing all of the tags). On a RegionTracker *miss* no information is available about the location of the block, so power and latency are increased as the conventional L2 tags have to be accessed subsequent to the RegionTracker access.

3.3.1 Relative Storage Requirements

RegionTracker can reduce power and latency provided that it is sufficiently smaller than the regular tag arrays, and that it captures a sufficiently large fraction of requests. Tables 2 and 3 report the storage requirements

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

RegionTracker	*L2*	*Power*	*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

(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 arrays of a 2Mbyte L2 cache. Of course, this is a first-order approximation of area cost. Considering the overall bit requirement is meaningful independent of whether the L2 tag array is partitioned into separate banks or sub-arrays since the CRH and the CBV can be similarly partitioned. Table 2 shows CRH requirements for entry counts of 512 through 4K. The size of each CRH entry is a function of the total number of L2 cache blocks and of L2's associativity. As shown in Table 2 even a 4K CRH requires less than 15% of the storage needed by the conventional tag array (in our evaluation we demonstrate that even a 512-entry CRH is sufficient for a 2Mbyte L2 cache). In general, the number of bits required by a CRH is *N x lg(L2 blocks + 1)* where

N is the number of CRH entries. This formula pessimistically assumes that it is possible for all L2 block to map onto the same CRH entry. This is possible for a fully associative L2, however, for lower associativities there are implicit restrictions that reduce the maximum counter values possible in the CRH and hence its cost.

Table 2. 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	1.9%
1K	3.8%
2K	7.2%
4K	14.4%

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. The larger the region, the more blocks it contains, and the more block information fields each CBV entry requires. The length of the physical address also impacts CBV size, but to a much lesser extent, as it only affects the size of the region tags. As shown in Table 3, 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 3 can also be used to estimate the relative cost of RegionTracker for larger caches since CBV requirements are not affected by cache size. For example, a 4Mbyte cache requires 1.923 times as many tag bits as a 2Mbyte cache. Accordingly, it follows that a 128-entry CBV with 8Kbyte regions requires just 4.5% of the bits needed by the L2 tag array. We include CBVs of up to 512 entries because they represent meaningful designs for caches larger than 2Mbytes. For example, the 512 entry CBV with 8K regions requires just 4.8% of the bits required by a conventional tag array for a 16Mbyte L2 cache with 128-byte blocks.

Table 3. Fully-associative CBV storage requirements (region tags and per block location information) 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%	4.1%	7.9%
64	<1%	1.0%	1.4%	2.4%	4.3%	8.1%	15.8%
128	1.5%	1.9%	2.9%	4.7%	8.6%	16.2%	31.6%
256	2.9%	3.8%	5.7%	9.5%	17.1%	32.5%	63.2%
512	5.9%	7.7%	11.4%	19.0%	34.3%	64.9%	126.3%

3.3.2 RegionTracker vs. a Tag Set Buffer

Most previously proposed techniques for reducing power and latency for lookups at a high level rely on caching a small subset of the tag information (see Section 4). In this section, we discuss how RegionTracker differs from a straightforward *tag set buffer*, or *TSB* for short such as those proposed in [31,33]. 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.

As our results show TSBs do not scale well enough for larger, higher-level caches. This is because, TSBs rely primarily on locality in the set stream. Beyond a point, not much set locality appears in the L2 reference stream, primarily because the L1 cache filters much of this locality. Relatively small TSBs capture most of the potential that exists, and, as we increase the number of TSB entries, we quickly reach an area of diminishing returns or the TSB becomes comparable in size to the L2 tag array. As demonstrated Section 5.2, RegionTracker scales well as more resources are devoted to it.

In addition, the access path in RegionScout quickly becomes shorter than that of a TSB. Specifically, accessing a direct-mapped TSB involves the following steps: (1) use the set part of the incoming address to index the TSB and compare it against the set tag associated with the corresponding tag buffer entry, then (2), read out the corresponding set entry, and finally (3), compare the tag portion of the address with the tags in the TSB entry. If the TSB is associative, then many of these steps occur simultaneously. In RegionTracker we simultaneously compare the region portion address with the region tags and access the corresponding block information for the tracked regions. The match signals then select the appropriate block information entry amongst those simultaneously read while the match was performed. For the same associativity, RegionTracker requires fewer comparisons and reads fewer bits compared to a TSB. RegionTracker also overlaps the two accesses (region tag match and block information access) whereas in a conventional TSB the two accesses are sequential (first we lookup the set tags and then read the corresponding tag set).

Finally, a TSB requires complete tag set contents. Accordingly, it has to be placed next to the tag structures it caches or additional large numbers of wires are needed to communicate complete tag sets from the tag arrays to the TSB. RegionTracker maintains way information which is collected one block at the time. This is particularly important for non-conventional cache hierarchies such as DNUCA [15] and NuRapid [10] that unify all higher-level caches into a larger structure with variable, distance-dependent latency. Both TSB and RegionTracker need to observe clean replacements.

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 [33], 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.,* [25] 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.

A number of techniques have also been proposed for reducing the area and power of tag arrays. Decoupled sectored caches [30] and Caching Address Tags [32] 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 impact indirectly 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. The implementation of these techniques requires changing the L2 cache controller and can be proven a lot more complex than that of the simple RegionTracker structures. Other techniques which address tag array power include way prediction [12,14,27] or memoization [20], as well as techniques which attempt to optimize tag search energy using multi-stage tag lookup [9,8]. The former techniques, as well as [4] and [24] apply mostly to the L1 instruction cache, while the latter techniques were demonstrated for the L1 data cache. It is not clear if they 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 [17, 19, 31, 18, 11].

Bloom filters like CRH have been previously proposed for avoiding snoop-induced tag lookups [22] or snoop broadcasts [23], for L1 hit/miss prediction [26], load/store queue complexity reduction [29] and for miss prediction [21]. 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 part of the paper is organized as follows: In Section 5.1 we describe our experimental methodology. In Section 5.2 we demonstrate that practical RegionTrackers can capture many L2 references, and we also show per program behavior. In Section 5.3 we report power 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 4. 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). 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 ran using a reference input data set. It was not possible to simulate some of the benchmarks due to insufficient memory resources. As a result the following SPEC CPU 2000 benchmarks are included our experiments: *ammp, applu, apsi, art, bzip2, crafty, eon, equake, facerec, fma3d, galgel, gcc, gzip, lucas, mesa, mgrid, parser, swim, twolf, vortex, vpr* and *wupwise*. To obtain reasonable simulation times, samples were taken for 10 billion committed instructions. In the measurements reported, we first skipped 100 billion instructions prior to collecting measurements for all benchmarks except for art and parser for which we only skipped 20 billion instructions. We selected this measurement interval after experimenting with several other 10 billion instruction samples and with longer samples of up to 40 billion instructions. We have 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. Table 5 reports the memory footprint per benchmark, showing that in most cases it exceeds by far the on-chip L2 cache capacity. Reasonably sized RegionTrackers could easily track most, if not all, blocks for an application with a very small memory footprint. For all experiments except for those in Section 5.5 we used functional simulation. For the studies in Section 5.5 we used timing simulation. The memory system comprises split, 2-way set-associative level one data and instruction caches of 32Kbytes each, with 64-byte blocks, a unified, 8-way set-associative L2 cache with 128-byte blocks and a main memory. We studied L2 caches in the range of 2Mbytes to 16Mbytes. The latencies are shown in Table 4. 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 that are presented we use an 8Kbyte region size. While we have seen that larger regions may be preferable (Section 2) the corresponding measurements assumed that regions form a continuous area of memory. Had we simulated a virtual memory system this property would be guaranteed only for regions that do not exceed the OS page size. Given that many modern OSes use 8K pages we restrict our attention to this region size. We note that virtual memory page allocation could affect RegionTracker for larger than a page regions either way. To estimate power and latency for the various structures we modified CACTI [28] to determine the optimal number of cache sub-arrays for a 0.10um process. To model the serialized L2 cache access, we optimized the access delay of the tag and data paths separately and modified the L2 tag array model to appropriately account for bitlines, wordlines and sense amps. Since RegionTracker uses simple SRAM structures we modified CACTI to model its power and latency also. To measure power dissipation at the architectural level, we used the Wattch framework [8] with the aforementioned power models.

Table 4. 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

5.2 Coverage with Practical Structures

This section demonstrates that practical RegionTrackers can capture many L2 references. We report the *coverage* exhibited by various RegionTrackers, that is the fraction of L2 accesses that find definite block location information from the RegionTracker alone. For clarity, we first study *average* coverage and then present per program measurements. Average coverage allows us to concisely present measurements over several configurations. Figure 3 reports average coverage for various fully-associative RegionTrackers and L2 capacities. Specifically, the number of CRH entries is varied from 512 to up to 4K as reported along the x-axis. Three different CBVs are considered with 64, 128 or 256 entries. Results are presented for 2Mbyte and 4Mbyte caches. Each curve corresponds to a combination of CBV and cache capacity and is labeled as *A-B* where A is the number of CBV entries and B the L2 capacity. Observed coverage varies from as low as 27% for the 512/64 RegionTracker and the 4Mbyte cache and as high as 76% for the 4K/256 RegionTracker with the 2Mbyte cache. Generally, coverage increases with the number of CRH and CBV entries. The increase in coverage is more pronounced in the range of 512 to 2K CRH entries. This result combined with the results of Figure 1 (where we assumed perfect first miss detection) suggest that a 2K CRH provides much of the coverage possible with an ideal CRH. Coverage appears to be less sensitive to CBV entry count. Much of the coverage can be obtained even with the 64-entry CBV. Depending on the cache capacity and the CRH entry count, doubling the number of CBV entries increases coverage anywhere between 5% to 9%. While we do not show additional results we note that further increasing the CBV entry count results in similar behavior. At approximately 2K CBV entries coverage nears 95% for all programs studied while it is very close to perfect for most. However, a 2K CBV is a relatively large structure (larger than the L2 tags for a 2M cache). These results show that practically sized RegionTrackers can capture many L2 references. For example, the 1K/128 RegionTracker offers coverage of 57% when used with a 2Mbyte cache. The amount of storage required by this RegionTracker is just 12.4% of that required by the conventional L2 tags.

Table 5. 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	*mesa*	10M
bzip2	188M	*mgrid*	57M
crafty	2M	*parser*	62M
eon	2M	*swim*	196M
equake	50M	*twolf*	3M
facerec	17M	*vortex*	70M
fma3d	107M	*vpr*	51M
galgel	45M	*wupwise*	181M

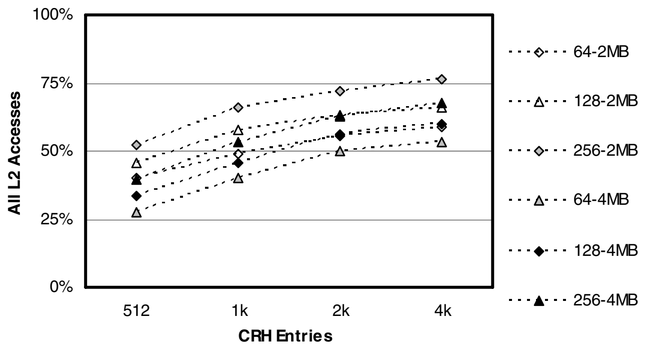
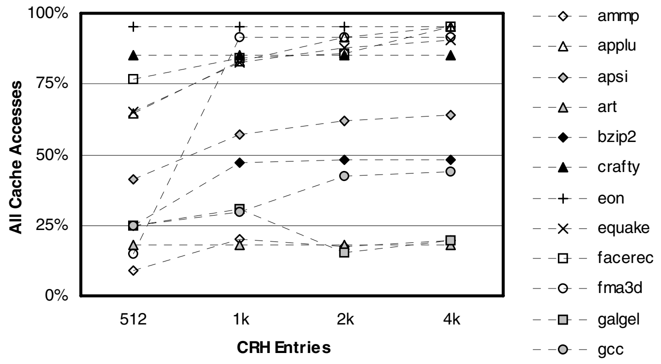


Figure 3: *Coverage (i.e., fraction of L2 references serviced by the RegionTracker) with practical RegionTrackers. Region size is 8K. The number of CRH entries is varied from 512 to up to 4K (x-axis). Three different fully-associative CBVs are shown with 64, 128 or 256 entries (curves). Finally, results for 2Mbyte and 4Mbyte caches are shown. Each curve corresponds to a different CBV and L2 cache combination (labelled as CRH_Entries-Cache_size).*

5.2.1 *Per Program Coverage*

Average behavior can be misleading. Accordingly, Figure 4 reports per program coverage for X/128 RegionTrackers where X varies as reported along the x-axis (range of 512 to 4K CRH entries). For clarity, we restrict our presentation to the 2Mbyte cache and split the results into two graphs. Per program varies greatly, exposing the trade-offs in tuning the CRH and the CBV. The larger the CRH, the more first misses are detected. However, as more first misses are detected, a larger CBV may be required to track more regions. In general, for all applications except vortex and galgel, coverage increases or remains constant as the number of CRH entries is increased. Coverage remains constant and is relatively high for crafty, eon and twolf. As reported in Table 5 the memory footprints of these applications are relatively small hence the RegionTracker can track most of blocks accessed for the application's lifetime. A small memory footprint is not a requirement for high coverage. For example, coverage is 93% with the smaller 512/64 RegionTracker for gzip, an application that has a 185Mbyte footprint. Observed coverage is below 25% for ammp, art, parser and vpr because they access more than 128 regions in close proximity in time. Accordingly, larger CBVs are needed to fine-track these regions. For example, coverage with a 256-entry CBV increases to 30%, 62%, 27% and 30% respectively. In vortex coverage is inversely proportional to the CRH entry count while in galgel a drop is observed going from 1K to 2K CRH entries. The larger CRHs detect more first misses and expose more active regions which in turn thrash the 128-entry CBV. Larger CBVs are needed for these two applications (512-entry for galgel and 256-entry for vortex).



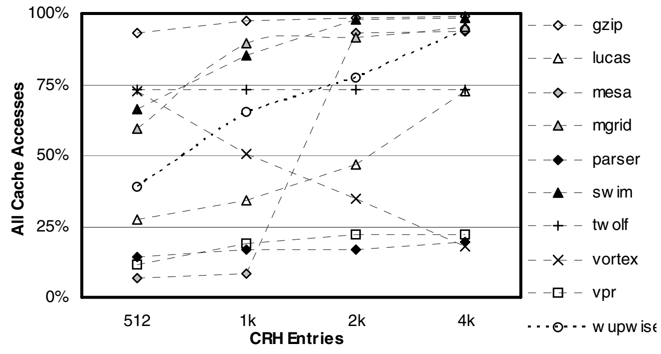


Figure 4: *Per program coverage for 8K regions, 2M L2 cache and 128-entry CBV for various CRH entry counts (x-axis).*

Due to limited space we do not report results with limited CBV associativity. However, in the rest of this paper we consider 8-way set-associative CBVs as their coverage is within 2% of that possible with a fully-associative CBV for the configurations we studied.

5.3 Power Savings Compared to Conventional L2 Tags

This section demonstrates that significant power savings are possible with practical RegionTrackers for various L2 caches. We consider L2 caches with sizes from 2Mbytes to up to 16Mbytes. Figure 5 reports average power savings reported as a fraction of the power dissipated by the conventional L2 tags. We report results for RegionTrackers that have either 64 or 128 entry CBVs and CRH entry counts of 512 to up to 4K (x-axis). Each combination of CBV entry count and cache size is reported on a different curve and is labeled as A-B where A is the number of CBV entries and B the L2 cache size. The highest power savings of 47% is observed for the 2K/128 RegionTracker and the 2Mbyte L2 cache (which is just 2% higher than that possible with the 1K/128 RegionTracker). This RegionTracker produces savings of 45%, 36% and 23% for the 4Mbyte, 8Mbyte and 16Mbyte caches respectively. Thus relatively high savings are possible for all cache capacities studied. The savings are reduced as the cache capacity is increased for the same RegionTracker. By increasing CBV entry count it is possible to increase coverage and power savings for the larger caches. As the cache capacity increases more regions remain cache resident hence fewer first misses occur. Accordingly, a larger CBV is needed to track them. 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.

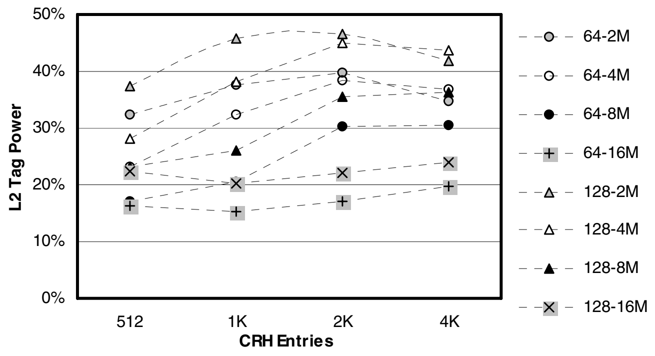


Figure 5: *Average power savings expressed as fraction over the power of a conventional L2 tag array. Shown are RegionTrackers with various CRH entry counts (x-axis) and a 64- or 128-entry CBV. Results are shown for L2 caches of 2Mbytes to up to 16Mbytes. Each curve corresponds to a different CBV and L2 configuration (labeled as CBV_Entries-L2_Capacity). RegionTrackers are 8-way set-associative.*

For completeness we report per program power changes for the best and the worst average cases which are, respectively, the 2Mbyte L2 cache with the 2K/128 RegionTracker and the 16Mbyte L2 cache with the 1K/64 RegionTracker. For the best average case, RegionTracker is robust producing power savings for all programs. For the worst average case, RegionTracker is not as robust as power may increase, albeit only by at most 1%. Accordingly, it is important to tune the RegionTracker configuration to match the underlying memory system. Fortunately, RegionTracker tuning is primarily influenced by L2 capacity.

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 TSBs of various sizes. We use average coverage for comparison. Figure 7 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. Parts (a) and (b) report coverage for a 2Mbyte and 4Mbyte caches respectively. The grey bars report coverage for TSBs of

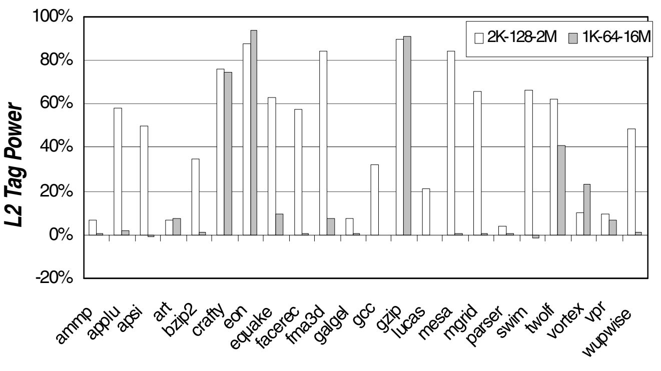


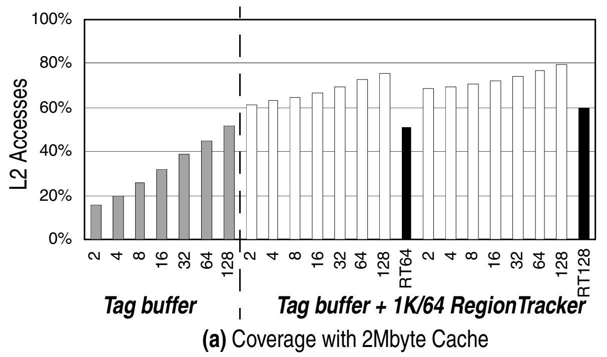
Figure 6: *Per program relative power change with RegionTracker expressed as a fraction over the conventional L2 tag array power. Results are shown for the best and worst average cases (as reported in figure 5). The best average power was measured for a 2Mbyte L2 cache with a 2K/128 CRH (white bars). The worst average case was measured for a 16Mbyte L2 cache with 1K/64 (grey bars). The RegionTrackers are 8-way set-associative.*

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 seven bars are for the 1K/64 RegionTracker and the next seven bars are for the 1K/128 RegionTracker. Finally, the two dark bars report coverage with just the 1K/64 (left) and the 1K/128 (right) RegionTrackers respectively. Table 6 reports the storage requirements in bits of the TSBs and the two RegionTrackers as a fraction of the L2 tags. This is a first-order approximation of the cost and power of these structures. As we explained in Section 3.3 the latency and power of RegionTrackers is lower than a TSB of similar size. Also note that each TSB entry contains a full set of eight tags plus a set index tag.

Coverage increases with the number of TSB entries. Still, the 1K/64 RegionTracker that is smaller offers coverage that is virtually identical to that of the 128-entry tag set buffer with the 2Mbyte cache and very close with the 4Mbyte cache. The combination of a RegionTracker and of a TSB outperforms all standalone configurations. More importantly, adding just a two entry TSB to a 1K/64 RegionTracker offers coverage at least 7% better than that possible with the 128-entry TSB. These results suggest that the two approaches are to some extent complimentary. There is some short-term set locality in the L2 stream that even a very small TSB can capture. RegionTracker on the other hand can capture most of the far-flung locality that exists in a more efficient manner.

5.5 Performance and Overall Power

RegionTracker affects L2 latency, and thus impacts both overall performance and power. We have measured the overall performance impact of a 1K/128 8-way set-associative RegionTracker, assuming that it decreases L2



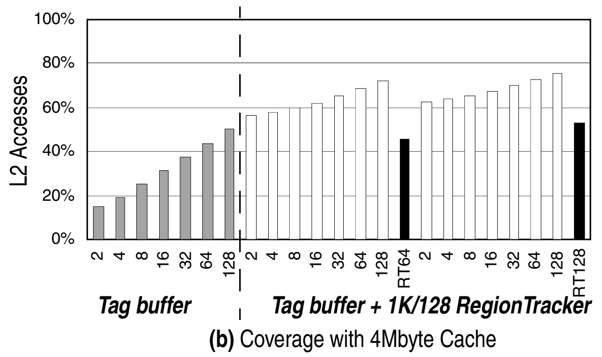


Figure 7: *Comparing Tag Buffers of various sizes to RegionTracker in terms of coverage. (a) 2Mbyte cache, and (b) 4M cache. The first seven grey bars correspond to the coverage of fully-associative tag buffers of two through 128 entries. The next seven white bars are for combined RegionTrackers with Tag Buffers with the number of entries reported along the x-axis (two through 128). The RegionTracker has a 64-entry CBV that is 8-way set-associative and a 1K-entry CRH. The coverage of the RegionTracker alone is shown by the next dark bar (RT64). Finally, we double the number of RegionTracker entries to 128.*

Table 6. 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	1.3%
32	2.6%
64	5.3%
128	10.6%
CBV 64 + 1K CRH	8.1%
CBV 128 + 1K CRH	12.4%

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 WATTCH [8]. We studied caches of 2Mbytes and 4Mbytes. On average, overall performance increased only slightly with RegionTracker, but it never decreased. In the best case of gzip performance increased by 6% with a 2Mbyte cache. Correspondingly, overall processor power always decreased slightly with the two RegionTracker configurations we studied.

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 1k/128 RegionTracker captures 56% of all L2 references, saving 38% of the tag lookup power for a 4Mbyte L2 cache, while requiring less than 5% 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

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 and funds from the University of Toronto.

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. *Architectural 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] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.