The Bunker Cache for Spatio-Value Approximation

Joshua San Miguel[†], Jorge Albericio[†], Natalie Enright Jerger[†] and Aamer Jaleel[‡]
[†]University of Toronto, [‡]NVIDIA
joshua.sanmiguel@mail.utoronto.ca, {jorge, enright}@ece.utoronto.ca, ajaleel@nvidia.com

***Abstract*—The cost of moving and storing data is still a fundamental concern for computer architects. Inefficient handling of data can be attributed to conventional architectures being oblivious to the nature of the values that these data bits carry. We observe the phenomenon of *spatio-value similarity*, where data elements that are approximately similar in value exhibit spatial regularity in memory. This is inherent to 1) the data values of real-world applications, and 2) the way we store data structures in memory. We propose the *Bunker Cache*, a design that maps similar data to the same cache storage location based solely on their memory address, sacrificing some application quality loss for greater efficiency. The Bunker Cache enables performance gains (ranging from 1.08× to 1.19×) via reduced cache misses and energy savings (ranging from 1.18× to 1.39×) via reduced off-chip memory accesses and lower cache storage requirements. The Bunker Cache requires only modest changes to cache indexing hardware, integrating easily into commodity systems.**

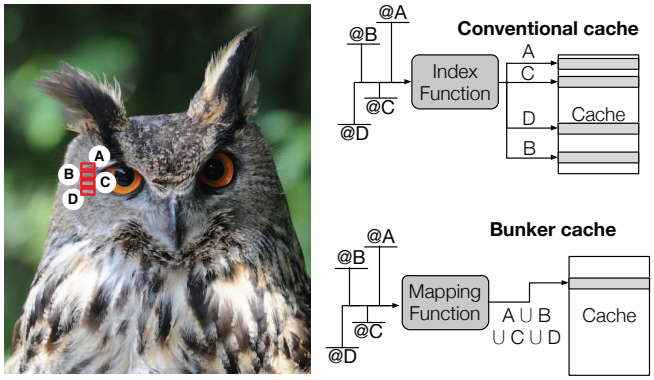


Fig. 1: Bunker Cache overview.

I. Introduction

As today's data sets grow to enormous proportions, the cost of moving and storing data on-chip continues to challenge computer architects. The latency and energy of an off-chip memory access is still several orders of magnitude greater than a simple arithmetic operation. As a result, we try to preserve as much data as possible on-chip at the expense of nearly half of the chip area dedicated to caches [19], [40]. A major source of inefficiency is that we architects too often treat data as nothing more than streams of bits when in reality, these bits encode real-world information and data values that applications need to process. And in the real world, such information is very often approximate and redundant in space and time. For example, images are discretized grids of repeating colors and sounds are noisy waves with periodicity in time.

This insight fits well in the wake of approximate computing [10], [13], [17], [21], [25], [33], [35], [39], a design paradigm that recognizes that many applications only need to meet acceptable quality levels and do not require the absolute precision that conventional architectures provide. Our work exploits the inherent approximate nature of data as opposed to remaining oblivious to the information that the bits carry. We identify and explore the phenomenon of *spatio-value similarity*: data elements that are approximately similar in value tend to be stored at regular intervals in memory. Consider the image in Figure 1. Neighbouring pixels are naturally similar in value. However, since images are stored in row-major order, adjacent pixels along a column are not spatially colocated in memory but rather stored at intervals of the row size.

We propose the *Bunker Cache*,[1] a design that exploits an application's spatio-value similarity to reduce both off-chip memory accesses and cache storage requirements. It does this by storing similar data blocks in the same location in the cache since substituting one for the other still yields acceptable application quality. Other value approximation techniques have been proposed recently [37], [44], [47], particularly Doppelgänger [36], which deduplicates approximately similar blocks to compress data. Unfortunately, these come at a cost of dramatic changes to hardware and significant complexity that arises from the need to manipulate data values themselves. On the other hand, spatio-value similarity is a consequence of the inherent nature of 1) the real-world information we process, and 2) the way we store this information in memory. This means that we can determine which data blocks are likely to be similar in value based solely on where they are located. The example in Figure 1 demonstrates this, where the addresses (not the values) of similar blocks `A`, `B`, `C` and `D` are used to store them in the same cache location. This allows us to save storage while avoiding the complexity of manipulating data values, which is expensive considering the wide variety of data types and value representations that need to be supported. It also reduces cache misses and off-chip memory accesses, which is possible since there is no need to retrieve the data values from memory. Furthermore, the crux of the Bunker Cache implementation is in its mapping function, leaving much of the cache architecture itself unchanged. This greatly eases its adoption in commodity processors.

[1]The Bunker Cache is named after the famous conjoined twin brothers Chang and Eng Bunker.

978-1-5090-3508-3/16/$31.00 ©2016 IEEE

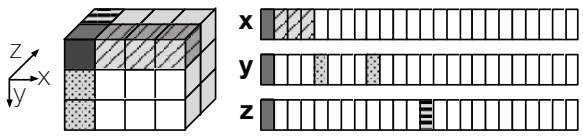


Fig. 2: Example of spatial regularity in memory. This shows how elements in a three-dimensional structure (left) are stored in one-dimensional memory (right).

We make the following novel contributions:

- We uncover significant spatio-value similarity in PERFECT [4] and AxBench [13] applications.
- We design and evaluate the Bunker Cache as a last-level cache, demonstrating considerable speedups (1.08× to 1.19×), dynamic energy savings (1.18× to 1.39×) and leakage power savings (1.18× to 1.47×) at only modest loss in application output quality.

II. Spatio-Value Similarity

Too often we architects treat data as a "black box"; merely a sequence of memory address requests that govern when and where to move bits. However, data is inherently some representation of real-world information. And real-world information is highly redundant in space and time, and variations in information are smooth and sparse. An image, for example, is simply a two-dimensional projection of a scene, and colors in scenes are typically clustered and vary in smooth gradients. A sound wave can be represented as a sparse set of frequencies or a continuous signal in the time domain with smooth and periodic repetitions. Neighbouring particles in a dense fluid are acted upon by similar external forces and tend to converge towards the same velocities. This inherent nature of real-world information is obfuscated and abstracted away as merely bits in a one-dimensional address space.

Though much of this information is ignored in conventional architectures, we observe that there is still regularity in the way data is stored. We characterize this via *spatio-value similarity*: similarity in data values at regular intervals in memory. Consider what happens to a general n-dimensional data structure when stored in one-dimensional memory. An example is shown in Figure 2. Given a three-dimensional data set, the figure shows how a row of elements in each dimension is mapped to memory. In this case, the x-dimension is the contiguous dimension; thus neighbouring elements in the x-dimension are stored together. Though neighbouring elements in the y- and z-dimensions are no longer spatially colocated, they are still stored at spatially regular intervals (or *strides*) in memory as shown by the shaded blocks in the figure. As discussed, in real-world data, neighbouring elements in n-dimensional space are typically similar in value. Our notion of spatio-value similarity captures this by recognizing the spatial regularity of how data is stored in memory.

To explore this, we examine approximate computing applications from PERFECT [4] and AxBench [13]. For each application, we take pairs of memory addresses that are some arbitrary *stride*[2] apart and map them together. That is to say, we force addresses that are a given stride apart to store their data in the same storage location; if there is value similarity between them, then there should be little to no impact on the quality of the application output. Consider a data set with block addresses `0x10`, `0x11`, `0x12`, `0x13`, `0x14` and `0x15` in memory. With a stride of 3, address `0x10` would share storage with `0x13`, as would addresses `0x11` and `0x14`, and addresses `0x12` and `0x15`. We ask the question: By how much is output quality affected when every third address in memory shares the same value? And which strides yield the highest output quality? We sweep varying strides and observe their impact on the application's final output. Figure 3 shows the results, using signal-to-noise ratio (SNR) as a metric for output quality relative to the baseline precise output (complete methodology can be found in Section IV). Absolute SNR values are not significant at this point; we only focus on the trends. There is a remarkable amount of regularity in the results. We make two key observations:

1) There are noticeable hills and valleys where quality is considerably higher and lower respectively.
2) These hills and valleys are periodic.

We say that spatio-value similarity is high if there is considerable regularity in where similar values are located in memory. For example in Figure 3a, 2dconv has hills at strides of 1, 240 and 480 (measured in 64-byte blocks). This implies that each data element is similar in value to those that are 1, 240 and 480 blocks away from it. Looking closer, we see that 2dconv processes two-dimensional images. Each image is stored in row-major order in memory (the x-dimension is contiguous), and each row is 240 blocks wide. Thus we can conclude that each data element is similar in value to its neighbours. There is *x-similarity* between neighbouring columns at strides of 1, and there is *y-similarity* between neighbouring rows at strides of 240. This holds for all the other applications as well; the periodic stride of their hills and valleys correspond to the width of their innermost x-dimension. Thus spatio-value similarity is commonplace in real-world applications. In this paper, we show how spatio-value similarity enables significant performance and energy improvements in modern cache architectures, with only modest changes to the hardware.

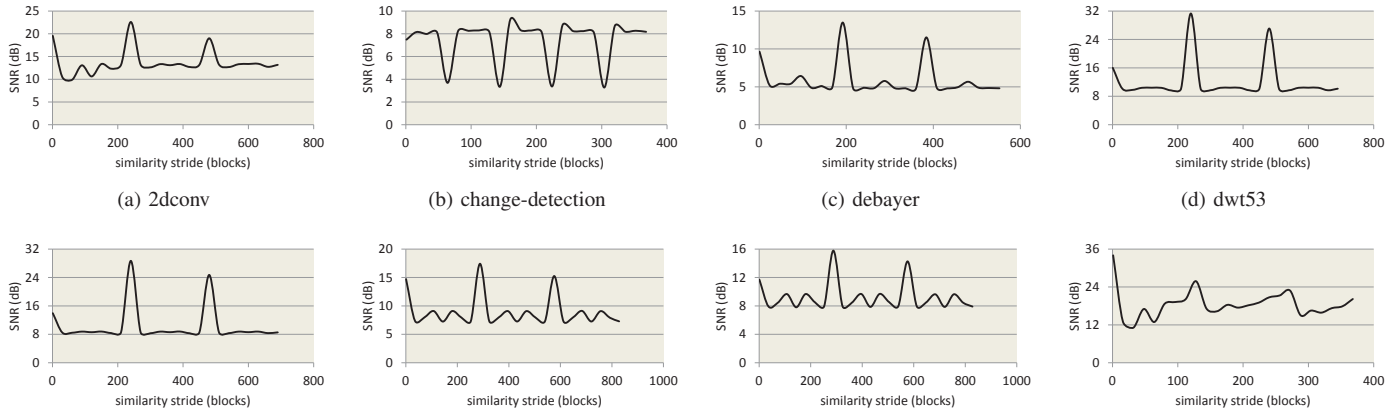
III. The Bunker Cache

By recognizing that approximate applications exhibit spatio-value similarity, we improve efficiency of accessing and storing data with mostly commodity cache hardware.

A. Overview

The Bunker Cache is a cache architecture that maps approximately similar blocks to the same cache location. It recognizes that such blocks can share a single storage location since replacing one with the other still yields acceptable application output. The Bunker Cache uses knowledge of the application's

[2]Strides are at the granularity of 64-byte block addresses.



(a) 2dconv (b) change-detection (c) debayer (d) dwt53

(e) histeq (f) jpeg (g) kmeans (h) lucas-kanade

Fig. 3: Application output quality with varying similarity strides.

spatio-value similarity to distinguish similar blocks. It only needs address information to identify similar blocks; it does not need to fetch the data values. This not only minimizes complexity but also allows for approximations to be made proactively. A miss for block `A` effectively becomes a hit if a similar block `B` is present in the cache. This enables performance gains via reduced cache misses as well as energy savings via reduced off-chip memory accesses and lower cache storage requirements. Furthermore, the implementation of the Bunker Cache can be contained entirely within the cache index function; we refer to this as the *spatio-value similarity mapping function*, or *mapping function* for short. Much of the cache architecture is left unchanged, easing the adoption of our technique in commodity systems.

The Bunker Cache mapping function (Section III-B) generates approximations by introducing a form of constructive aliasing. Instead of a conventional one-to-one mapping of a physical address to a cache set index plus a tag, we employ a many-to-one mapping. The mapping function associates addresses of multiple similar blocks to generate a single `bunkaddress`, allowing them to share the same entries in the tag and data arrays. It is important to note that the `bunkaddress` encompasses both the tag value and set index when performing a lookup in the Bunker Cache.

For much of the discussion, we assume that the application's spatio-value similarity is programmer- or user-specified. That is to say, the programmer or user selects a similarity stride (or strides) that they deem acceptable, and this stride is static for the entire execution. Later, we discuss mechanisms for dynamically adjusting the similarity mapping function for quality control (Section III-D2).

Since much of the architecture is unchanged, Bunker Cache operation (Section III-C) is similar to that of a conventional cache. We only require that a separate directory structure is used instead of embedding coherence information in the tag array. Coherence state and dirty bits must be tracked uniquely per physical address and thus cannot be stored in the Bunker Cache. Using separate directory structures is commonplace when using non-inclusive caches [11], [16] or non-traditional coherence protocols [48], [50].

B. Mapping Function

The mapping function is the crux of the Bunker Cache design. It performs the novel spatio-value similarity mapping: similar blocks are identified by their physical address and are mapped to the same `bunkaddress`. This section describes the mapping function implementation in detail.

1) Spatio-Value Similarity: To characterize spatio-value similarity in an application, the Bunker Cache mapping function defines two parameters: the similarity *stride* and *radix*. As discussed in Section II, the stride is the distance between blocks that are deemed acceptably similar. The radix defines the degree of similarity (i.e., the aggressiveness of the similarity mapping). For example, a radix of 2 implies that the Bunker Cache mapping function employs a two-to-one mapping of physical addresses to `bunkaddresses`. The radix can either be statically programmer- or user-specified or dynamically selected via quality control mechanisms (Section III-D2). As we show later in Section V, with higher radix comes more opportunity for efficiency gains but also more risk of output quality loss.

Figure 4 shows the algorithm and implementation of the Bunker Cache mapping function. It takes in the physical address as input and returns the `bunkaddress` as output. The mapping function also takes in the similarity stride (`STRIDE`) and radix (`RADIX`[3]). The similarity *window* (`WINDOW` = `RADIX` × `STRIDE`) is a derived parameter that represents the size of each contiguous chunk of memory in which similar blocks are mapped. Figure 5 shows an example where `STRIDE` is 3 and `RADIX` is 2. The address space is effectively divided into windows of 6 contiguous blocks; and within each window, every 3rd block is mapped together.

[3]`RADIX` does not have to be an integer; for example, `RADIX` 1.5 yields an equal mixture of `RADIX` 1 and `RADIX` 2 mappings.

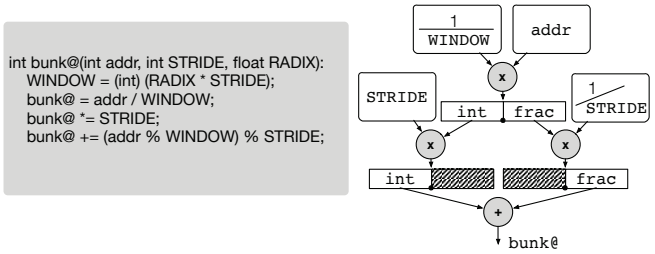


Fig. 4: Bunker Cache mapping function for associating addresses of similar blocks to the same `bunkaddress`.

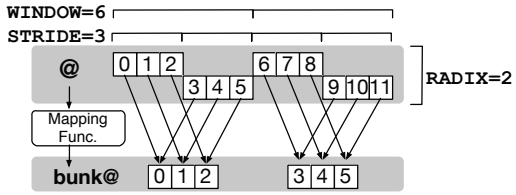


Fig. 5: Bunker Cache mapping function example. Addresses `0` to `11` are mapped to `bunkaddresses` using a stride of 3 and radix of 2.

`STRIDE`, `RADIX` and `WINDOW` (and their reciprocals) are assumed to be static during an application's execution; in our experiments, we find that all approximate data structures are static in size. Thus the expensive divide operations to compute reciprocals only need to happen once at the beginning of the application's execution. As Figure 4 shows, the mapping function uses only three fixed-point multiplications and one addition. Our implementation is cheap in terms of latency and cost, which is crucial since the mapping function is on the critical path of cache requests.

The mapping function effectively quantizes the address space by `WINDOW` (first step) and normalizes it to `STRIDE` (second step). The first step multiplies the input address with the reciprocal of `WINDOW`. This yields the integer and fractional parts of the product:

$$\texttt{int} = \lfloor \texttt{addr} \div \texttt{WINDOW} \rfloor$$
$$\texttt{frac} = \texttt{addr} \bmod \texttt{WINDOW}$$

In the second step, the `bunkaddress` is computed as:

$$\texttt{bunk@} = (\texttt{int} \times \texttt{STRIDE}) + (\texttt{frac} \bmod \texttt{STRIDE})$$

The Bunker Cache mapping function generalizes to spatio-value similarity in arbitrary n-dimensional data sets. If similarity exists along n dimensions, then it is a simple matter of invoking n iterations of the mapping function. For the example in Figure 2, if spatio-value similarity exists along all three dimensions, then the mapping function can be configured to iterate three times: first using the x-similarity stride of 1, followed by the y-similarity stride of 4, and finally the z-similarity stride of 12.

2) Approximate vs. Precise: Only memory addresses of approximate data are amenable to our similarity mapping. As we discuss later in Section IV-A, the specification of a block—approximate or precise—can be passed (via ISA extensions) with each request to the Bunker Cache. The mapping function can partition the cache to dedicate some sets for precise data. However, to minimize impact on performance, the cache must be partitioned to match the application's working set footprint, which can be challenging especially if the fraction of precise data varies over time. Instead we simply dedicate a bit in the `bunkaddress` that specifies whether or not a block is approximate or precise. This is more flexible to the application's working set. Precise requests bypass the similarity mapping and simply use the physical address as the `bunkaddress`, indexing into the cache as in conventional systems. Similarly, disabling the mapping function simply involves setting `RADIX` to 1; in which case, the mapping logic is power-gated and bypassed completely.

C. Cache Operation

Since much of the design is contained within the mapping function, the Bunker Cache does not introduce considerable changes in operation. In this section, without loss of generality, we describe its operation as a shared last-level cache in a chip multiprocessor.

1) Lookups: Upon lookup, the physical address of a cache request is passed through our mapping function. The set index and the tag from the resulting `bunkaddress` are used to locate the corresponding block in the Bunker Cache. This is identical to a lookup in a conventional cache, with the exception that a single `bunkaddress` can correspond to multiple addresses.

The Bunker Cache architecture is oblivious to this extra mapping step and operates as if the `bunkaddress` is the actual memory address. We effectively emulate an environment where if blocks A and B are similar, they are treated as if they are a single block. In parallel, a regular lookup is performed in the directory. The directory lookup bypasses the Bunker Cache mapping function since coherence and dirty state are tracked uniquely per physical address. Thus the directory has the same effective capacity as the baseline.

2) Insertions and Replacements: Upon a miss in the Bunker Cache, a request (using the physical address, not the `bunkaddress`) is sent to memory to fetch the data block. In the meantime, a new entry is allocated both in the cache (using the `bunkaddress`) and in the directory (using the memory address). If the miss triggers a replacement, the victim cache block is deallocated. In order to issue writebacks and back-invalidations (for inclusive caches), we need to perform lookups for addresses in the directory that correspond to this victim block. This requires performing the Bunker Cache mapping function backwards. Fortunately, this is off the critical path of cache requests. The backward mapping function is similar to Figure 4, except instances of `STRIDE` and `WINDOW` are swapped. The backward function is one-to-many; it yields `RADIX` lookups in the directory to retrieve coherence and

dirty state. The overheads of these lookups are modeled in our energy numbers. In our experiments, much of the approximate footprint is read-only, so overheads are low.

Though we opt for a design with minimal changes to the cache architecture, an alternative solution is to store multiple sets of coherence and dirty bits in the Bunker Cache. In this way, no additional directory structure is needed. Note that an eviction in the directory does not provoke an eviction in the last-level cache; blocks can reside in the cache even without tracking coherence and dirty state (though writebacks are provoked if the block is dirty). A clean block can remain in the cache since no private caches are currently using it; any subsequent requests for the block simply reallocate a directory entry and send the data from the cache instead of memory.

3) Writes and Coherence: Write requests and writebacks to the Bunker Cache operate similarly to conventional caches. As with all lookups, the cache is accessed via our mapping function while the directory is accessed using the physical address. Coherence state and dirty bits are updated accordingly, while the data block is updated in the case of a writeback. Since coherence is still tracked using memory addresses, the coherence protocol is unaffected in the rest of the cache hierarchy. Coherence and dirty bits can alternatively be replicated in the Bunker Cache to avoid the need for a separate directory; we simply opt for a design that makes use of commodity cache hardware. Since our mapping is many-to-one, a writeback from one address affects the data values of others that share the same storage location. This can also lead to incoherence between private copies of approximate data. Such writebacks are tolerable since these addresses are expected to be similar in value anyway, and incoherence is tolerable since the application is naturally resilient to approximations on its data. These scenarios are modelled in our experiments; we find that output quality is not severely degraded.

D. Cache Optimizations

The previous sections describe the core Bunker Cache design, which can be implemented with mostly commodity cache hardware, only requiring changes to the index function. In this section, we discuss design enhancements that may require modifications to the cache architecture, though at a modest cost.

1) Drowsy Blocks: With our many-to-one mapping function, an application's cache footprint is effectively compressed by a factor of `RADIX`. This implies that the Bunker Cache has lower storage requirements than a conventional cache. To exploit this, unused blocks in the Bunker Cache can be put into a drowsy state [14] to save leakage power. Recall that the directory is indexed by the physical address, so it has the same effective capacity as the baseline. We consider a block in the Bunker Cache as unused if there are currently no entries in the directory that map to it. Such a block has been evicted from the directory and is no longer used elsewhere in the cache hierarchy. Thus it can be put into a drowsy state. To enable drowsy states in the Bunker Cache, an additional $log_2(\texttt{MAX\_RADIX}) + 1$ bits are needed in each tag entry to keep a reference count of how many entries in the directory are currently mapped to it. A count of zero implies an unused block, which can then be set to drowsy state.

2) Dynamic Quality Control: The similarity radix acts as a design knob for tuning the aggressiveness of our approximation technique. A higher radix yields more similar mappings, resulting in higher efficiency gains but more quality loss. At times, it is desirable to tune the radix dynamically, particularly to target some user-specified quality level. In our evaluations, we use SNR as a metric for quality.

To enable dynamic quality control in the Bunker Cache, the user specifies a target quality level $\texttt{SNR}_\texttt{t}$. The Bunker Cache splits the application runtime into epochs, measured in approximate LLC accesses (100,000 in our experiments). Within each epoch, approximate accesses to the cache are sampled (100 samples in our experiments) and accumulated into a running SNR value for the epoch: $\texttt{SNR}_\texttt{e}$. To demonstrate how this works, consider an access to physical address `A` is selected as a sample. Regardless of whether or not the access is a hit, an off-chip memory request is triggered to fetch `A`. In addition, we also send a memory request for the block at address $\texttt{A} + \texttt{WINDOW}/2$. Upon receiving both data blocks, we compute the SNR between the data elements in `A` and $\texttt{A} + \texttt{WINDOW}/2$, effectively performing a quality check on their actual values from memory. The data type and size of each element can be retrieved via program annotations and ISA extensions (Section IV-A). This SNR value is then accumulated into $\texttt{SNR}_\texttt{e}$. At the end of each epoch, we compare against $\texttt{SNR}_\texttt{t}$ and update `RADIX` accordingly:

$$\texttt{if SNR}_\texttt{e} > (1+\alpha)\texttt{SNR}_\texttt{t}\texttt{, RADIX := RADIX} \times \beta$$
$$\texttt{if SNR}_\texttt{e} < (1-\alpha)\texttt{SNR}_\texttt{t}\texttt{, RADIX := RADIX} \div \beta$$

where α and β are design parameters (0.5 and 4 in our experiments respectively). If this comparison triggers a change in `RADIX`, then all approximate data is flushed from the cache. All overheads associated with this are modelled in our results. Thus for each epoch, our dynamic quality control mechanism effectively asks: Is the quality of the current radix too high or too low relative to the target? If $\texttt{SNR}_\texttt{e}$ is close enough to $\texttt{SNR}_\texttt{t}$, then the current radix is sufficient.

If desired, the similarity stride can also be tuned dynamically using the same mechanism. Instead of tuning `RADIX`, various `STRIDE` values are swept in a geometric fashion (e.g., `STRIDE` of 1, 2, 4, 8, etc.). This is inspired from branch prediction where geometric histories are used to find the optimal history length efficiently. However, as we explain later in Section IV-A, the appropriate stride for a given application can most often be determined statically.

IV. Methodology

In this section, we describe the benchmarks and experimental methodology we use in our evaluations.

A. Benchmarks and Program Annotations

We perform experiments on benchmarks from PERFECT [4]—2dconv, change-detection, debayer, dwt53, histeq,

2dconv	240	change-detection	128
debayer	192	dwt53	240
histeq	240	jpeg	288
kmeans	288	lucas-kanade	128

TABLE I: Strides (in blocks) used for y-similarity.

lucas-kanade—and AxBench [13]—jpeg, kmeans—which are suites for embedded computing and approximate computing, respectively. We select these benchmarks because they are common in image processing (the killer app for the Bunker Cache) so that we are able to visualize their outputs for the purposes of evaluating quality, with the exception of lucas-kanade whose output matrix is more complex to visually render. It is important to note that these benchmarks have broader applicability beyond image processing: 2dconv in linear algebra, dwt53 in communications, histeq in data mining, kmeans in machine learning. All benchmarks are parallelized via pthreads and configured for large input sets.

Similar to prior work, we assume that approximate data is explicitly annotated in the program. This is facilitated by programming languages and tools for approximate computing [3], [5], [34]. We annotate our benchmarks by hand, identifying large data structures that are amenable to approximation.[4] This information is then passed down to the architecture via simple ISA extensions [12]. Alternatively, such information can be inferred dynamically via monitoring and profiling mechanisms to identify which data is amenable to approximation, though this is more challenging. The design of the Bunker Cache can support any means of specifying approximate data. As explained in Section III-A, we also assume that spatio-value similarity information is programmer- or user-specified. This can be passed down to the architecture using the same mechanisms and only needs to be done once in the beginning.

Our benchmarks use two-dimensional data structures; we explore x-similarity and y-similarity in our evaluations (recall Section II). The x-dimension is contiguous in memory and thus corresponds to a stride of 1 (i.e., neighbouring elements in the x-dimension are 1 block apart). Table I shows the y-similarity strides that we use in our evaluations. The strides are based on the width of each row in the contiguous x-dimension of our data sets. For example, in 2dconv, neighbouring elements in the y-dimension are 240 blocks apart. Note that our selected strides correspond to the period of hills in Figure 3. Selecting an appropriate stride is straightforward since it is only dependent on the dimensions and structure of data sets, which are typically static and independent of the data values. Thus in most cases, the appropriate stride can be declared via program annotations.

B. Experiments and Evaluation Metrics

We implement the Bunker Cache as a shared last-level cache (LLC) in a four-core chip multiprocessor, configured as in Table II. We evaluate it in terms of application output quality, dynamic energy savings, leakage power savings and performance—application speedup and reduction in misses-per-kilo-instructions (MPKI)—relative to a baseline conventional cache. We also compare our design to prior work on cache compression—base-delta-immediate (BΔI) [30] and uniDoppelgänger [36]—in terms of cache storage savings.

[4]In our experiments, we assume large superpages of 256 MB for these data structures so that similarity strides do not cross physical page boundaries, preventing 1) similarity mappings between non-contiguous virtual pages and 2) access violations between different processes. This is applicable since in our benchmarks, all approximate data structures are static in size and allocated only once in the beginning.

processor	4 IA-32 cores, 1 GHz, 4-wide OoO, 80-entry ROB
private L1	16 KB, 4-way, LRU, 1-cycle, 64 B blocks
private L2	128 KB, 8-way, LRU, 3-cycle
shared LLC	2 MB, 1-bank, 16-way, LRU, inclusive, 6-cycle (13-cycle Bunker)
directory	2 MB-equivalent (2 K entries), 1-bank, 16-way, LRU, 6-cycle
main memory	1 GB, 160-cycle latency
cache coherence	MSI protocol

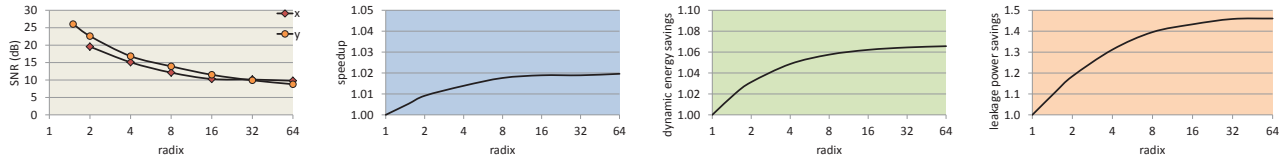
TABLE II: System configuration used in evaluation.

We run simulations using Pin [23] to measure output quality and a full-system cycle-level simulator [28] to measure runtime. Pin simulations are run to completion while full-system simulations are capped at 100 M user-mode instructions. We use signal-to-noise ratio (SNR) as a proxy for quality loss between the approximate and precise baseline outputs. For images, SNR measures how much of the original precise image is maintained after employing our approximation technique. It is a standard metric in image processing and is typically measured in decibels (dB), where ∞ dB corresponds to zero quality loss. Note that the outputs of our benchmarks are often used for further processing and not for their visual aesthetic. Output quality is a subjective metric and highly depends on the use case of an application. For this reason, we present our results as tradeoff curves and show sample outputs of our benchmarks in our evaluations, for both precise and approximate executions. This allows readers to visually grasp the impact of our technique and make their own judgments on the quality-efficiency tradeoffs.

We estimate dynamic energy of all caches and the directory using CACTI [26] and assume 640 pJ per 32-bit off-chip DRAM access [18]. For the Bunker Cache index function, we assume 0.1 pJ and 3.1 pJ for 32-bit integer add and multiply operations respectively [18]. We also use CACTI to estimate the leakage power of the total on-chip cache hierarchy including the directory. For cache blocks in drowsy state, we scale leakage according to the energy parameters provided by Flautner et al. [14]. We assume 0.9 pJ and 3.7 pJ for 32-bit floating-point add and multiply operations respectively for our dynamic quality control mechanism [18].

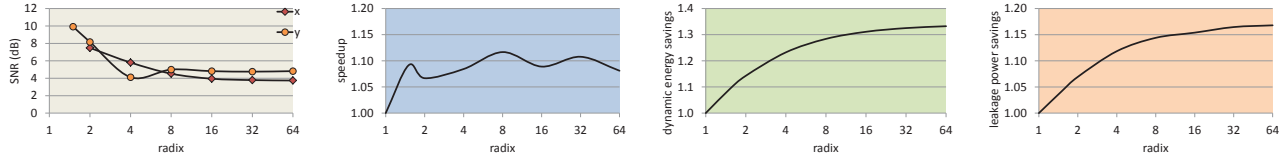
V. Evaluation

We present the quality-efficiency tradeoffs of our design on each of our benchmarks. We also compare against prior work on cache compression and evaluate the Bunker Cache when equipped with further optimizations.



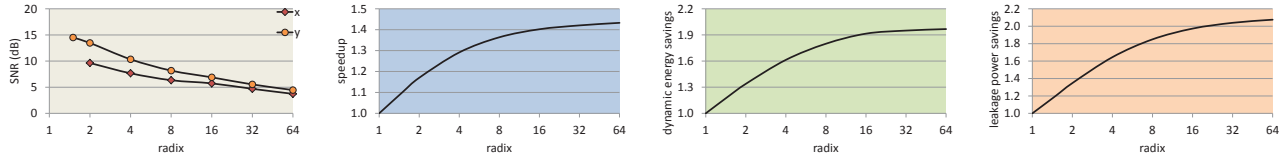
(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 6: Evaluation of 2dconv: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.



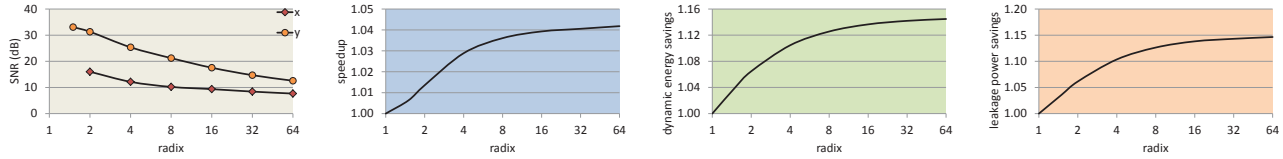
(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 7: Evaluation of change-detection: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.



(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 8: Evaluation of debayer: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.



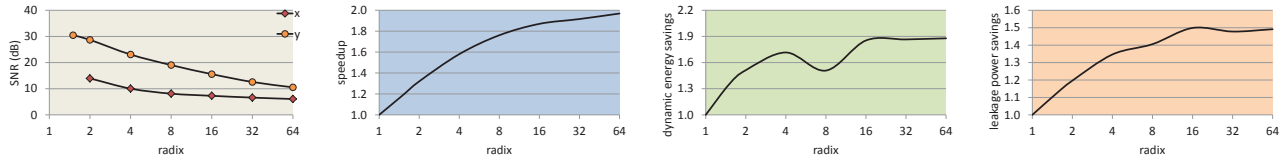
(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 9: Evaluation of dwt53: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.

A. Quality-Efficiency Tradeoffs

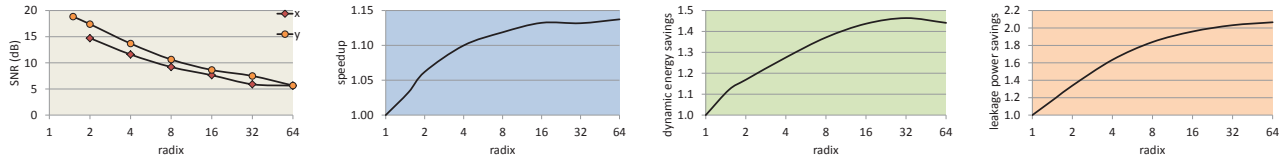
Figures 6-13 show the quality-efficiency tradeoffs per benchmark. In this section, we evaluate application output quality, speedup and dynamic energy savings as a function of the spatio-value similarity radix. Dynamic energy pertains to the total on-chip cache hierarchy (including the directory) as well as off-chip memory. We also present leakage power savings when using drowsy cache blocks, though we discuss these results later in Section V-C. We provide example output images of our benchmarks in Figures 15-21 to visualize the impact of our approximation technique. We evaluate trends and present our results in this manner so that the reader can also make their own judgments on the tradeoffs, since quality is an inherently subjective metric.

Output Quality. As described in Section III-B1, the similarity radix effectively acts as a design knob for the degree (i.e., aggressiveness) of our approximation technique. The universal trend in Figures 6a-13a is that output quality degrades as radix increases. This is expected since a higher radix yields more memory addresses mapping to a bunkaddress. The amount of quality loss varies depending on the benchmark. Computations that perform large reductions of approximate data tend to be more tolerant to aggressive approximations. This is because application output quality is based on the final reduced value;



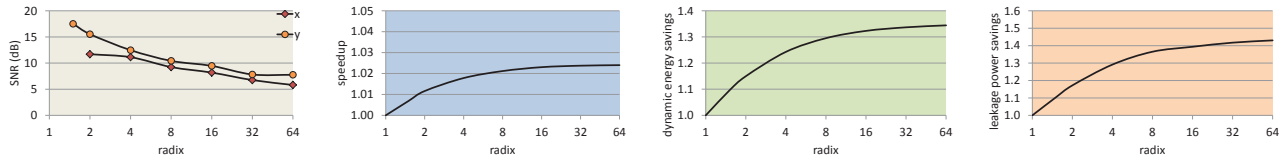
(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 10: Evaluation of histeq: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.



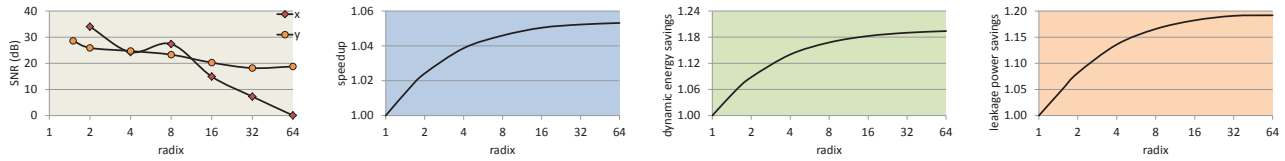
(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 11: Evaluation of jpeg: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.



(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 12: Evaluation of kmeans: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.



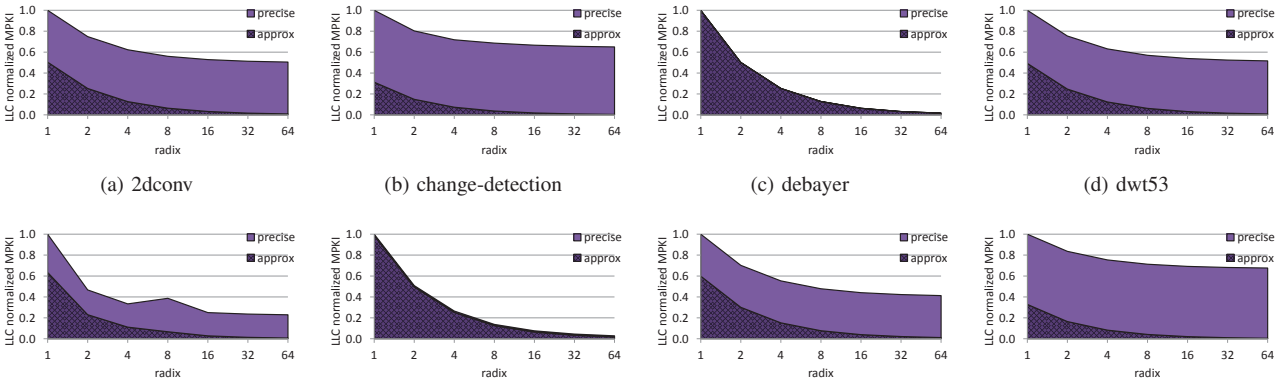
(a) output quality (b) application speedup (c) dynamic savings (d) leakage savings

Fig. 13: Evaluation of lucas-kanade: output quality (x- and y-similarity) and speedup, dynamic energy savings and leakage power savings (y-similarity) with varying radix.

any quality loss in approximating an individual data element is weighted less significantly when measuring quality loss in terms of the final reduced value. For example, histeq (Figure 10a) constructs a normalized histogram of its entire input data set; quality loss of a single element becomes less significant since the final histogram aggregates all elements together, obfuscating small variations in the quality of individual elements.

To get a better sense of quality loss per benchmark, Figures 15-21 show example outputs, comparing the baseline precise output against the Bunker Cache. Though we only show outputs for y-similarity radix of 4, it is important to note that outputs at higher radix values are not much worse in quality and are very likely to still be acceptable. We only discuss trends in quality and not absolute SNR values since output quality is highly subjective. For example, change-detection (Figure 7a) yields the lowest average output quality of all benchmarks, yet it adequately detects image changes as shown in Figure 18. The low SNR values are primarily due to the binary nature of the output; for each pixel location, the output is either 0 or 1. Thus any pixel that returns the wrong value technically ends up with worst-case quality loss.

Figures 6a-13a compare the quality trends of x-similarity and y-similarity. On average, y-similarity delivers higher output quality for our benchmarks. The reason for this is that our



(a) 2dconv (b) change-detection (c) debayer (d) dwt53

(e) histeq (f) jpeg (g) kmeans (h) lucas-kanade

Fig. 14: Normalized LLC MPKI for precise and approximate data with varying y-similarity radix.

(a) baseline precise

(a) baseline precise

(a) baseline precise

(b) y-similarity radix 4

Fig. 15: Output of 2dconv.

(b) y-similarity radix 4

Fig. 16: Output of dwt53.

(b) y-similarity radix 4

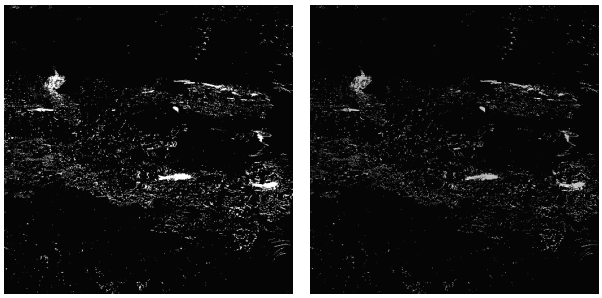
Fig. 17: Output of histeq.

similarity mapping is performed at a cache block granularity. Cache blocks are fairly large (64 B), so they can contain several (in some cases, many) data elements. This negatively impacts x-similarity since the x-dimension is the contiguous dimension. For example, consider a 4-byte pixel `P` at location `[i][j]`. We expect that it is approximately similar in value to its neighbour in the x-dimension at `[i][j+1]`. However, its neighbour is likely to be within the same cache block as `P`. Since there are 16 pixels per block, employing the x-similarity mapping (i.e., stride of 1) makes `P` equivalent to `[i][j+16]` instead, which is more likely to degrade quality. On the other hand, y-similarity yields higher output quality since it is not the contiguous dimension. This means that `P` can be mapped to `[i+1][j]` given the appropriate stride.

For the rest of our evaluation, we focus only on y-similarity. On average, output quality SNR varies from 18.8 dB, 12.5 dB and 8.2 dB for y-similarity radix of 2, 8 and 64 respectively. For reference, this yields 3.7%, 7.7% and 13.4% respectively in terms of normalized root-mean-square error, another common metric in approximate computing [13].

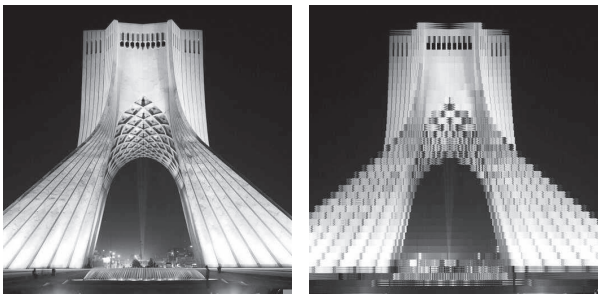
Efficiency. As Figures 6b-13b and Figures 6c-13c show, application speedup and dynamic energy savings increase with the radix: on average, 1.08× speedup and 1.18× energy savings with radix 2, and 1.19× speedup and 1.39× energy savings with radix 64. Both are primarily due to significant reductions in LLC misses, which are shown in Figure 14. As radix increases, mapping more memory addresses to a single cache location yields more hits in the Bunker Cache. This mitigates both the latency and energy cost of accessing off-chip memory. This is a benefit that prior work on cache deduplication [36], [46] cannot obtain since they must fetch the actual data values in order to perform the deduplication, whereas the Bunker Cache only needs memory address information. Note that this benefit applies even for compulsory misses in the LLC.

The efficiency gains vary per benchmark, depending on the application's sensitivity to the LLC and the distribution of



(a) baseline precise (b) y-similarity radix 4

Fig. 18: Output of change-detection.

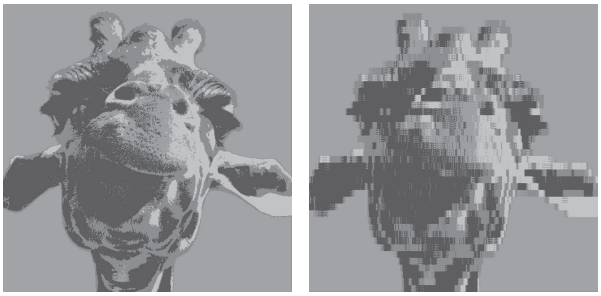


(a) baseline precise (b) y-similarity radix 4

Fig. 20: Output of jpeg.

(a) baseline precise (b) y-similarity radix 4

Fig. 19: Output of debayer.



(a) baseline precise (b) y-similarity radix 4

Fig. 21: Output of kmeans.

approximate and precise LLC misses, shown in Figure 14. In general, a higher approximate MPKI corresponds to greater speedup and dynamic energy savings. Of our benchmarks, debayer, histeq and jpeg achieve the highest MPKI reduction since they have the largest fraction of approximate data in their working sets, benefiting most from our similarity mapping. This yields speedups of 1.29×, 1.58× and 1.10× with radix 4. Though jpeg has a larger approximate footprint than histeq, histeq sees higher speedups since jpeg is more computation-heavy and less memory-bound (baseline LLC MPKI of 0.92) compared to histeq (baseline LLC MPKI of 5.43). In our experiments, we find that the energy cost of off-chip memory is more significant than its latency cost, generally resulting in higher energy savings than performance gains. These three benchmarks save 1.61×, 1.72× and 1.28× of dynamic energy with radix 4 and nearly 2× at even higher radix values.

B. Cache Compression

The Bunker Cache offers a form of cache compression with minimal changes to the cache architecture (i.e., only the mapping function). Figure 22 shows compression ratio (geometric mean) with varying y-similarity radix. As expected, compression ratio increases with radix, offering more opportunity for LLC storage savings. As with the speedup and dynamic energy savings, the extent to which storage is saved depends on how much of the footprint is approximate.

Figure 22 compares the compression ratio of our technique against BΔI [30] and uniDoppelgänger [36] (with 14-bit similarity maps). BΔI is a state-of-the-art intra-block compression technique that operates on both approximate and precise data. The uniDoppelgänger Cache is an inter-block deduplication technique that captures value similarity for only approximate data. At low radix, the Bunker Cache has a lower compression ratio than BΔI, primarily since BΔI is applied to both precise and approximate data whereas our technique is only for approximate data. However, the approximate footprint is relatively large, and approximate data are far more amenable to compression due to applications' tolerance to quality loss. As a result, the Bunker Cache eventually achieves higher compression ratio at higher radix (i.e., beyond radix of 4). The uniDoppelgänger Cache achieves a high compression ratio since it is able to find similar blocks at arbitrary memory locations. This comes at a high complexity cost whereas the Bunker Cache can use mostly commodity cache hardware. Furthermore, our technique only needs memory addresses to identify similar blocks whereas uniDoppelgänger needs to fetch the actual data values. This enables us to reduce LLC misses and off-chip accesses in addition to storage compression. It is important to note that BΔI, uniDoppelgänger and our technique are all orthogonal; they can be applied simultaneously for even more storage savings.

C. Cache Optimizations

We evaluate the Bunker Cache with additional optimizations of drowsy blocks and dynamic quality control.

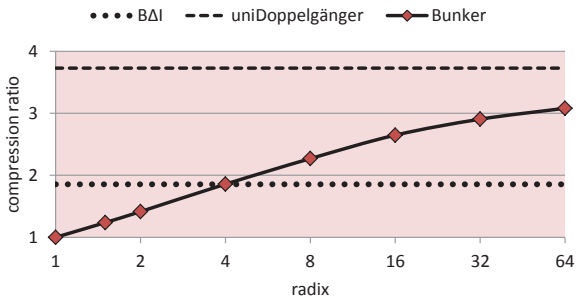


Fig. 22: Compression ratio with varying y-similarity radix, compared to BΔI and uniDoppelgänger.

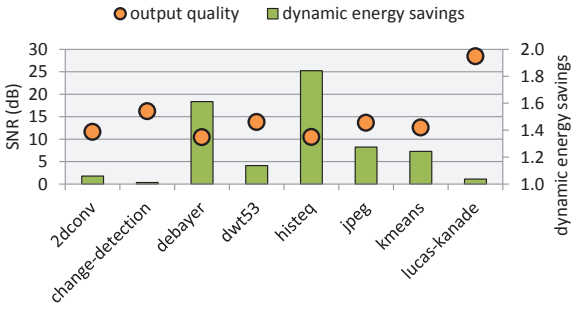


Fig. 23: Output quality and dynamic energy savings using dynamic quality control of y-similarity radix. Initial radix is 4 and target SNR is 10 dB.

Drowsy Blocks. As discussed in Section V-B, spatio-value similarity enables storage savings. This can translate to leakage power savings by enabling drowsy state for cache blocks, as described in Section III-D1. Figures 6d-13d show the leakage power savings with varying y-similarity radix when using drowsy blocks. Leakage power pertains to the total on-chip cache hierarchy including the directory. These results account for the additional reference counter bits to track unused cache blocks. As expected, more leakage power is saved with higher radix, and benchmarks with larger approximate footprints achieve more savings (e.g., debayer and jpeg approach 2× at higher radix). Note that leakage power can alternatively be saved by reducing the size of the LLC according to the average compression ratio, which is a typical approach in cache compression work.

Dynamic Quality Control. We implement our dynamic quality control mechanism for the Bunker Cache, described in Section III-D2. In our experiments, we assume epochs of 100,000 approximate cache accesses. We sample 100 of these accesses per epoch to compute $\texttt{SNR}_\texttt{e}$. We set the design parameters α and β to 0.5 and 4 respectively, derived empirically.

Figure 23 shows how our dynamic quality control mechanism impacts both output quality and dynamic energy savings for each benchmark. For these experiments, we select a target SNR value of 10 dB and address only y-similarity. The initial radix is set to 4; the minimum and maximum are 1 (i.e., precise case) and 64 respectively. The results show that our mechanism is able to keep output quality near the target 10 dB consistently in all benchmarks. Though we introduce overheads in flushing the cache and computing SNR values, we find that for our benchmarks, the radix tends to settle early: radix of 1 for change-detection and lucas-kanade; 4 for debayer, jpeg and kmeans; 16 for 2dconv; and 64 for dwt53 and histeq. This implies that there are only a few changes in radix (and thus cache flushes) during execution, allowing us to achieve high dynamic energy savings despite the overheads (up to 1.61× and 1.84× for debayer and histeq). Given the large epochs and infrequent sampling, the cost of our dynamic quality checks is outweighed by the significant reduction in off-chip memory accesses.

VI. Related Work

This section discusses how the Bunker Cache relates to previous proposals on cache hash functions and value locality.

Cache Hash Functions. Instead of simply using some of the block address bits as the cache index, previous research has proposed better hash functions [20], which have been implemented in commercial processors [1]. Skew-caches [43] use a different hashing function per way, reducing the number of conflict misses but breaking the concept of a set. Inspired by cuckoo hashing [29], a technique originally proposed to create space-efficient hash tables, the ZCache [40] provides associativity by increasing the number of replacement candidates but not the number of ways. The Bunker Cache implements a mapping function which effectively groups together data with similar values which are spatially close in the data-structure space.

Value Locality. Value locality was first explored in the context of value prediction [6]–[9], [15], [22], [27], [41]; current values for instructions can be predicted based on previous values. Value prediction faces implementation challenges due to rollbacks, pipeline complexity and memory consistency models [24]. Recent work has revisited practical and efficient implementation techniques [31], [32]. To avoid many of these complexities, one can relax the notion that predicted values must match exactly with actual values [37], [38], [44], [45], [47]; this observation marries well with approximate computing. Value redundancy is exploited in other techniques such as cache compression [2], [30], frequent value caching [49], address correlation [42] and cache deduplication [46]. Extending the latter, Doppelgänger [36] exploits approximate value similarity but at the cost of a radically new and more complex cache architecture. The Bunker Cache exploits a fundamentally different locality known as spatio-value locality and does so with minimal changes to the cache structure.

VII. Conclusion

In this paper, we explore the notion of spatio-value similarity in approximate computing. Such similarity exists in general applications whose data sets exhibit value locality between neighbouring elements. We propose the Bunker Cache, an architecture that maps approximately similar blocks to the same

cache location using knowledge of an application's spatio-value characteristics. This effectively treats similar blocks as one and the same; a miss for block A is now a hit if a similar block B is present in the cache. This enables significant reductions in cache misses, off-chip memory accesses and cache storage requirements. Furthermore, the implementation of the Bunker Cache is contained entirely within the cache index function. Thus we offer an approximate computing technique that can achieve efficiency gains with mostly commodity hardware. Our experiments uncover significant spatio-value similarity in image processing applications; however, Bunker Cache's applicability is not limited to such applications. Many of the applications we consider have uses beyond image processing; further exploration of the application space is left for future work.

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their thoughtful feedback. This work is supported by a Queen Elizabeth II/Montrose Werry Scholarship in Science and Technology, the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, the Ministry of Research and Innovation Early Researcher Award and the University of Toronto.

REFERENCES

[1] "UltraSPARC T2 supplement to the Ultra- SPARC architecture 2007," Sun Microsystems, Tech. Report, 2007.
[2] A. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *International Symposium on Computer Architecture*, 2004.
[3] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.
[4] K. Barker *et al.*, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013, http://hpc.pnnl.gov/projects/PERFECT/.
[5] J. Bornholt *et al.*, "Uncertain<T>: A first-order type for uncertain data," in *ASPLOS*, 2014.
[6] M. Burtscher, "Improving context-based load value prediction," Ph.D. dissertation, University of Colorado, 2000.
[7] M. Burtscher and B. G. Zorn, "Load value prediction using prediction outcome histories," University of Colorado, CS Department Technical Report CU-CS-873-98, 1998.
[8] M. Burtscher and B. G. Zorn, "Exploring last n value prediction," in *PACT*, 1999.
[9] B. Calder *et al.*, "Selective value prediction," in *Proc. Int. Symp. Computer Architecture*, 1999.
[10] V. K. Chippa *et al.*, "Analysis and characterization of inherent application resilience for approximate computing," in *Proc. Int. Design Automation Conference*, 2013.
[11] P. Conway *et al.*, "Cache hierarchy and memory subsystem of the AMD opteron processor," *IEEE Micro*, 2010.
[12] H. Esmaeilzadeh *et al.*, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.
[13] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.
[14] K. Flautner *et al.*, "Drowsy caches: simple techniques for reducing leakage power," in *ISCA*, 2002.
[15] F. Gabbay, "Speculative execution based on value prediction," Technion - Israel Institute of Technology, EE Department Technical Report 1080, 1996.
[16] J. Gaur *et al.*, "Bypass and Insertion Algorithms for Exclusive Last-level Caches," in *ISCA*, 2011.
[17] J. Han and M. Orshansky, "Approximate computing: an emerging paradigm for energy-efficient design," in *Proc. European Test Symposium*, 2013.
[18] S. Han *et al.*, "Learning both Weights and Connections for Efficient Neural Networks," 2015, arXiv:1506.02626 [cs.NE].
[19] D. Kadjo *et al.*, "Power gating with block migration in chip-multiprocessor last-level caches," in *ICCD*, 2013.
[20] M. Kharbutli *et al.*, "Using prime numbers for cache indexing to eliminate conflict misses," in *HPCA*, 2004.
[21] D. S. Khudia *et al.*, "Rumba: An Online Quality Management System for Approximate Computing," in *Proc. Int. Symp. Computer Architecture*, 2015.
[22] M. H. Lipasti *et al.*, "Value locality and load value prediction," in *ASPLOS*, 1996.
[23] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[24] M. M. K. Martin *et al.*, "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing," in *MICRO*, 2001.
[25] T. Moreau *et al.*, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *HPCA*, 2015.
[26] N. Muralimanohar *et al.*, "CACTI 6.0: A Tool to Model Large Caches," Technical Report HPL-2009-85, HP Labs, 2009.
[27] T. Nakra *et al.*, "Global context-based value prediction," in *HPCA*, 1999.
[28] N. Neelakantam *et al.*, "FeS2: a full-system execution-driven simulator for x86," poster presented at ASPLOS, 2008.
[29] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.
[30] G. Pekhimenko *et al.*, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *PACT*, 2012.
[31] A. Perais and A. Seznec, "EOLE: Paving the way for an effective implementation of value prediction," in *Proc. of the International Symposium on Computer Architecture*, 2014.
[32] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *HPCA*, 2014.
[33] M. Samadi *et al.*, "SAGE: Self-tuning approximation for graphics engines," in *MICRO*, 2013.
[34] A. Sampson *et al.*, "EnerJ: approximate data types for safe and general low-power consumption," in *PLDI*, 2011.
[35] A. Sampson *et al.*, "Approximate storage in solid-state memories," in *Proc. Int. Symp. Microarchitecture*, 2013.
[36] J. San Miguel *et al.*, "Doppelganger: A cache for approximate computing," in *MICRO*, 2015.
[37] J. San Miguel *et al.*, "Load value approximation," in *International Symposium on Microarchitecture*, 2014.
[38] J. San Miguel and N. Enright Jerger, "Load value approximation: Approaching the ideal memory access latency," in *Workshop on Approximate Computing Across the System Stack*, 2014.
[39] J. San Miguel and N. Enright Jerger, "The anytime automaton," in *ISCA*, 2016.
[40] D. Sanchez and C. Kozyrakis, "The ZCache: decoupling ways and associativity," in *MICRO*, 2010.
[41] Y. Sazeides and J. Smith, "The predictability of data values," in *Proc. Int. Symp. Microarchitecture*, 1997.
[42] R. Sendag *et al.*, "Address correlation: Exceeding the limits of locality," *IEEE Computer Architecture Letters*, 2003.
[43] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. Int. Symp. Computer Architecture*, 1993.
[44] M. Sutherland *et al.*, "Texture cache approximation on GPUs," in *WAX*, 2015.
[45] B. Thwaites *et al.*, "Rollback-free value prediction with approximate loads," poster presented at PACT, 2014.
[46] Y. Tian *et al.*, "Last-level cache deduplication," in *ICS*, 2014.
[47] A. Yazdanbakhsh *et al.*, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *ACM TACO*, vol. 12, no. 4, 2016.
[48] J. Zebchuk *et al.*, "A Tagless Coherence Directory," in *Proc. Int. Symp. Microarchitecture*, 2009.
[49] Y. Zhang *et al.*, "Frequent value locality and value-centric data cache design," *ACM SIGOPS Operating Systems Review*, vol. 34, pp. 150–159, 2000.
[50] H. Zhao *et al.*, "SPATL: Honey, I Shrunk the Coherence Directory," in *PACT*, 2011.