Byte-Select Compression

MATTHEW TOMEI, University of Illinois at Urbana-Champaign

SHOMIT DAS and MOHAMMAD SEYEDZADEH, Advanced Micro Devices, Inc.

PHILIP BEDOUKIAN, Cornell University

BRADFORD BECKMANN, Advanced Micro Devices, Inc.

RAKESH KUMAR, University of Illinois at Urbana-Champaign

DAVID WOOD, Advanced Micro Devices, Inc., University of Wisconsin-Madison

Cache-block compression is a highly effective technique for both reducing accesses to lower levels in the memory hierarchy (cache compression) and minimizing data transfers (link compression). While many effective cache-block compression algorithms have been proposed, the design of these algorithms is largely ad hoc and manual and relies on human recognition of patterns. In this article, we take an entirely different approach. We introduce a class of “byte-select” compression algorithms, as well as an automated methodology for generating compression algorithms in this class. We argue that, based on upper bounds within the class, the study of this class of byte-select algorithms has potential to yield algorithms with better performance than existing cache-block compression algorithms. The upper bound we establish on the compression ratio is 2X that of any existing algorithm. We then offer a generalized representation of a subset of byte-select compression algorithms and search through the resulting space guided by a set of training data traces. Using this automated process, we find efficient and effective algorithms for various hardware applications. We find that the resulting algorithms exploit novel patterns that can inform future algorithm designs. The generated byte-select algorithms are evaluated against a separate set of traces and evaluations show that Byte-Select has a 23% higher compression ratio on average. While no previous algorithm performs best for all our data sets which include CPU and GPU applications, our generated algorithms do. Using an automated hardware generator for these algorithms, we show that their decompression and compression latency is one and two cycles respectively, much lower than any existing algorithm with a competitive compression ratio.

CCS Concepts: • **Hardware** → **Communication hardware, interfaces and storage**; • **Computer systems organization** → *Architectures*;

Additional Key Words and Phrases: Byte-select, cache compression, memory compression

ACM Reference format:

Matthew Tomei, Shomit Das, Mohammad Seyedzadeh, Philip Bedoukian, Bradford Beckmann, Rakesh Kumar, and David Wood. 2021. Byte-Select Compression. *ACM Trans. Archit. Code Optim.* 18, 4, Article 49 (September 2021), 27 pages.

<https://doi.org/10.1145/3462209>

Authors' addresses: M. Tomei and R. Kumar, University of Illinois at Urbana-Champaign; emails: tomei2@illinois.edu, rakeshk@illinois.edu; S. Das, M. Seyedzadeh, and B. Beckmann, Advanced Micro Devices, Inc.; emails: Shomit.Das@amd.com, Mohammad.Seyedzadeh@amd.com, Brad.Beckmann@amd.com; P. Bedoukian, Cornell University; email: pbb59@cornell.edu; D. Wood, Advanced Micro Devices, Inc., University of Wisconsin-Madison; email: David.Wood@amd.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1544-3566/2021/09-ART49

<https://doi.org/10.1145/3462209>

1 INTRODUCTION

Cache-block compression underlies cache [1–13], memory [14–19], and link compression [16, 20, 21] schemes. By compressing the information in a fixed-size cache block into a smaller representation (e.g., 64-byte blocks to 32 bytes), cache-block compression can effectively increase link bandwidth and the capacity of caches and memory. Prior work has demonstrated that a good cache-block compression algorithm enables significant improvements in system performance and energy efficiency. In this article, instead of focusing on any specific application of compression techniques, we focus on the broader *data block compression* challenge, since the algorithms are general to any use case that requires compressing small blocks of data.

While many cache-block compression algorithms have been proposed, their design has largely involved ad hoc selection of common patterns the designers see in data [1, 3, 4, 7, 21]. While these ad hoc algorithms have proven quite effective, they suffer from two primary drawbacks. First, it is challenging to identify practical upper bounds on compression ratios for given data. Information theoretic bounds on compression and algorithms that meet them exist, but these are not applicable to cache-block compression because of the small size of cache blocks being compressed and the random access support of compressed caches. Second, it is difficult to search the space of all possible cache-block compression algorithms. We solve these problems by focusing on byte-level dictionary-based compression. A large and powerful class of cache-block compression algorithms, including variants of many previously proposed compression algorithms, can be considered instances of the class of algorithms based on *byte selection*. In this context, we define byte selection as the process of choosing which bytes in an uncompressed cache block should be stored in the compressed block representation and how the bytes in the compressed representation are used to recreate the original uncompressed block.

One key benefit of defining the byte-select class is that limits on how well a byte-select algorithm can compress a given cache block are simple to define. An example of this is shown in Figure 1. The ideal byte-select algorithm will store only one copy of each unique byte. We can use this observation to bound the performance of byte-select algorithms. We show that there is a significant gap between previously proposed algorithms (even ones not in the byte-select class) and these bounds. This suggests that there may be realizable byte-select compression algorithms that perform better than any previously proposed algorithm.

Even after limiting our search to only byte-select algorithms, the space of algorithms is still extremely large. The naïve versions of byte-select algorithms that only store unique bytes require too much metadata, limiting the byte dictionary size and compression performance. Therefore, an intelligent method of searching for the most useful algorithms is required, as well as a way of generating efficient hardware for the chosen algorithms.

When generating new algorithms, we constrain the search space of algorithms further by describing them as a set of patterns that each apply to an entire cache-block. Each pattern in the set describes a way for a whole cache-block worth of bytes to be deduplicated (sufficient for compression and decompression). An example of such a pattern is given on the right side of Figure 1. It shows the pattern required to achieve the byte dictionary shown in the middle of the figure. The amount of metadata and complexity of algorithms can be controlled by limiting the number of patterns included in the algorithm. With this knob, we can trade off area and latency for compression ratio. Furthermore, with as few as 256 patterns, we can achieve competitive compression ratios and single-cycle compression and decompression latencies.

By automating the generation of compression algorithms, this work overcomes a key disadvantage of the previous intuition-driven approaches that rely on manual recognition of exploitable patterns. We find that our search method can find patterns that are well known to cache-block

Fig. 1. Using byte selection to compress lines. Here, three 4-byte pointer-like values are compressed into 4 bytes.

compression algorithm designers, as well as some new patterns, which we attribute our improved performance to.

In summary, this paper makes the following contributions:

- We establish upper bounds on compression ratios for byte-level dictionary-based algorithms. We find that byte-select algorithms have the potential to compress much more than previously published algorithms.
- We automate the generation of new cache-block compression algorithms. We set up the process as a search problem over patterns describing complete cache blocks in order to limit the complexity of the search. We solve the search problem using a mix of heuristics and optimality guarantees. We demonstrate the utility of our search strategy by successfully beating state-of-the-art algorithms for two different sets of quantized compressed sizes.
- We compare cache compression algorithms generated using our automated approach with existing algorithms, finding, on average, 23% higher compression ratio compared to the state-of-the-art. We describe some interesting differences from existing algorithms that may be carried into future algorithm designs.
- We generate RTL for the algorithms we generate and show that they have competitive area and have much lower latency (single cycle) than any existing competitive algorithm. We can generate a single algorithm that performs well on CPU and GPU applications, saving on design and validation effort.

2 BACKGROUND AND RELATED WORK

Cache-block compression is useful wherever data values exhibit common patterns and systems benefit from compressing those cache blocks. For instance, substantial research exists on increasing effective cache capacity by compacting compressed blocks into the cache data array. These compaction techniques include decoupled compressed caches [5] and skewed compressed caches [11].

Cache-block compression has also been shown to be useful for increasing link bandwidth [16, 20, 21]. For example, when the network datapath is narrower than a full cache block (e.g., two flits

used to transmit a 64-byte cache block), a block that can be compressed into half its original size can be transmitted in a single flit. This improves link bandwidth and reduces resource contention. Note that, unlike cache compression where random accesses among compressed cache-blocks will be made, it is possible to benefit from cross-cache-block temporal value locality when performing link compression. We do not target this type of locality in this work.

Many previous works have proposed cache-block compression algorithms. Alameldeen and Wood presented **Frequent Pattern Compression** [7] (FPC), which treats a cache block as a collection of 4-byte words and uses significance-based techniques to exploit spatial value locality (i.e., zero or sign-extending small values). **Cache Packer** [3] (CPack) builds on FPC by adding a dynamically updated dictionary of 4-byte words; CPack exploits temporal value locality by combining dictionary entries with 1-byte or 2-byte immediate values. Other algorithms focus on efficiently encoding zeros, which can account for 40% or more of the data values in some programs. **Zero-content augmentation** (ZCA) [13] is one such algorithm. FPC, CPack, and ZCA can all be implemented with byte muxes controlled by metadata. Whether they count as byte-select algorithms depends on whether you allow constant byte values in the byte dictionary that are not explicitly stored in the compressed format (used in all three algorithms) and whether sign checking falls under byte comparison (used in FPC). Our algorithm generation process does not allow constant byte values.

Other compression algorithms use addition or other operations to combine values in the dictionary. This cannot be implemented with byte selection without allowing arbitrary constant values in the dictionary. One such algorithm is **Base-Delta-Immediate** [4] (BDI), which uses delta encoding to encode 8, 4, or 2-byte words in the uncompressed block as a base plus a signed delta. **HyComp** [8] is a meta-compression algorithm, using data prediction techniques to select the best algorithm out of an established collection for a particular data set. **Bit-plane Compression** (BPC) [21] uses bit-planes to achieve more compression than other algorithms assuming data semantics can be described at a byte granularity, which is not true for floating point data. Our generated algorithm is competitive or beats BPC for all benchmark sets we evaluated (Section 7.1), showing this sub-byte level awareness is not necessary.

SC² [1] statistically analyzes the cache contents and uses a cross-cache-block Huffman coding to represent information. Other recent works also exploit cross-cache-block locality [6, 22]. Our work is limited to exploiting intra-cache-block locality. However, the hardware complexity benefits of byte selection are still applicable, and the search algorithm we propose can be applied to any trace of byte arrays. This includes byte arrays that contain multiple cache-blocks worth of data.

3 COMPRESSION AS BYTE SELECTION

In this section, we define and introduce byte select compression algorithms. We also demonstrate their utility by improving an existing algorithm using the constraint of byte selection.

3.1 Byte-select Basics

Byte-select compression attempts to map an uncompressed input block to a smaller, compressed representation consisting of a collection of bytes (called the *byte dictionary*) plus some metadata. Byte-select decompression reverses this process. Byte selection refers to the fundamental action underlying both compression and decompression. During compression, it refers to the process of selecting a subset of bytes in the uncompressed block and determining where to store them in the compressed representation's byte dictionary. Decompression selects which byte in the byte dictionary maps to each byte in the uncompressed output block. The compressed representation's metadata describes the mapping between the bytes in the byte dictionary and the uncompressed representation. This metadata is generated by the compression algorithm and used as an input to the decompression algorithm.

Fig. 2. Byte-select example.

Byte-select compression algorithms can be implemented using comparators, multiplexers, and random logic. The comparators determine which bytes are the same, multiplexors steer unique bytes (and potentially some duplicates) to the byte dictionary, and random logic controls the multiplexors and generates the metadata. Similarly, byte-select decompression uses random logic to decode the metadata and control multiplexors that route bytes from the byte dictionary to the uncompressed block.

Figure 2 shows a simplified example of byte-select compression and decompression. The 12-byte uncompressed input block is shown at the top of the figure and contains three 4-byte pointer-like values in this example. The compression algorithm uses comparators and random logic (not shown) to control the multiplexors (shown above the byte dictionary) to select an input byte for each byte in the compressed representation's byte dictionary. The compression algorithm also generates the metadata that describes the contents of the byte dictionary. In this example, the byte dictionary contains only the five unique bytes in the block (and only shows the selected inputs to the multiplexors). While storing only unique bytes obviously minimizes the dictionary's size (discussed further in Section 4.2), realizable algorithms frequently allow duplicate bytes to reduce the required metadata and the number of multiplexors.

The lower half of Figure 2 illustrates decompression, where multiplexors select which byte in the byte dictionary should be mapped to each byte in the decompressed output block. The metadata bits are used to generate the multiplexors' selector inputs either using random logic, or as illustrated in Figure 2, by indexing a metadata lookup table. Like the above, this example only shows the multiplexor inputs selected to decompress this specific block.

The large number of possible mappings between the uncompressed block and the byte dictionary give byte selection its expressive power. However, the area complexity of the byte-select compression and decompression hardware is determined by the number of byte inputs to the multiplexors and the complexity of the logic for generating and decoding metadata. The latency, which is a key constraint in cache data block decompression, is determined by the maximum number of inputs to any multiplexor and the depth of the metadata logic.

3.2 BSI

Conventional compression algorithms often use operations that attempt to maximize the efficacy of compression at the expense of latency and area. The byte-select class of algorithms, on the other

Table 1. BSI uses Byte Selection (i.e., base with selected bytes) while BDI uses Addition

	BDI	BSI
Area (μm^2)	5,634	1,820
Latency (ps)	407	223
Compression Ratio	1.35	1.36

BSI Performs as well as BDI and Requires Less Area and Less Latency.

Fig. 3. Byte selecting implementation of BDI.

hand, avoids using operations that can be expensive in terms of latency and area (even if they may be useful for compression). In fact, we believe that this cost-consciousness is responsible for the low latency of our generated algorithms (Section 7.2). A reformulation of an existing compression algorithm into a byte selection-based algorithm should also yield benefits. In order to demonstrate this, we transformed BDI into **BSI (Byte-Select Immediate)**, a derivative of BDI that uses byte selection rather than addition to combine the base and delta (i.e., the base provides the high-order bytes and the “delta” the low-order bytes). An example of this is shown in Figure 3. The three most significant bytes of the two 4-byte words in the 8-byte cache line are sourced from the same location in the dictionary, which is followed by the two unique deltas. There is a constant zero value separate from the byte dictionary which base values can be sourced from.

The primary intuition behind the algorithm remains the same: assume all words in the cache block are within a delta-sized region that includes the base. Byte selection in this case can be thought of as zeroing out the bits in the base value that overlap with the delta and then doing an unsigned addition of the delta. Byte selection is not too far off from the original signed addition to the unzeroed bits in terms of what is compressible. With a 1-byte delta for example, we will still be able to compress 256 values around the base value. Those values are no longer guaranteed to be evenly distributed on both sides of the base, but this turns out to not impact the compression ratio.

Table 1 shows that the compression ratios (with no quantization applied) for BSI and BDI. The ratio presented is the mean of means for the CPU, GPU Game, and GPU Compute trace sets described in Section 7.1. This methodology is used for all compression ratios in this article where the benchmark set is not explicitly stated. The compression ratios are almost identical (1.36 and 1.35, respectively). The advantage of BSI over BDI is that multiplexors are smaller and faster than adders. Our changes reduce the latency by nearly a factor of two and the area by over a factor of three.

4 MEASURES AND BOUNDS

In this section we introduce and justify a new metric, quantized compression ratio, and then discuss a set of bounds for byte-select algorithms.

4.1 Quantized Compression Ratio

Compression is only useful when we can utilize the space we free up. To limit hardware complexity, cache-block compression schemes typically target a small number of quantized sizes. For example, cache compression schemes typically compress a 64-byte block into a smaller number of bytes, but then must allocate 64, 32, 16, or 8 bytes regardless of the actual compressed size. Similarly, link compression schemes typically map a cache block to an integral number of flits. Thus, for flits

with a 32-byte payload, compression is only useful when a 64-byte cache block is compressed to no more than 32 bytes. Furthermore, compressing to less than the quantized size provides no benefit (unless it compresses to another, smaller quantized size).

Quantization has a significant impact of the efficacy of a compression algorithm, which is commonly measured using the *unquantized* compression ratio. The unquantized compression ratio is simply the original size divided by the compressed size, with both measured in bits or bytes. However, since quantization results in significant internal fragmentation, a more representative measure is the *quantized* compression ratio, where the compressed size is rounded up to the smallest quantized size that will be allocated. For example, FPC can represent a block of all zeros with 6 bytes, but in the link compression example above, we must still transmit a 32-byte flit. Thus the 32-byte quantized compression ratio more accurately represents the efficacy of this algorithm for this application.

Figure 4 illustrates how quantization impacts the compression ratio on average over all the workloads we studied (Section 7.1). BDI has the best unquantized compression ratio, 1.6, but this drops to 1.35 when quantized to the smallest of 32, 16, or 8 bytes, and to 1.16 when quantized to only 32 bytes. Furthermore, both BPC and CPack outperform BDI for both quantization sizes, i.e., quantization can change which algorithm appears best. For this reason, the rest of this paper will use quantized compression ratios.

4.2 Byte-Select Bounds

One of the benefits of the byte-select compression framework is that it is easy to identify a bound on the performance of the best possible byte-select algorithm. While undiscovered algorithms that are not in the byte-select class may indeed perform better than the best byte-select algorithm, this bound shows study of the byte-select class may lead to significant improvements over previously published algorithms. Since this bound exists within the constraints of the byte-select class with its in-built complexity constraints, we hope to be able to achieve a compression ratio close to the bound at low complexity.

The bound is simple: given a dictionary of size n , a cache block is considered compressible if it contains n or fewer unique bytes. We can set n to 32 (continuing our example) and assume an oracle provides the metadata. Alternatively, we can obtain a somewhat tighter bound by reserving one or more bytes for metadata and reducing n accordingly. In either case, this bound does not imply that a practical algorithm exists, but rather that no byte-select algorithm can do better given the same set of constraints.

The rightmost bars (labeled UB-M1) in Figure 4 illustrate the upper bound for byte-select compression algorithms with one byte of metadata. We show in later sections that a single byte of metadata is sufficient for practical and efficient algorithms. These bars show that byte-select compression algorithms have the *potential* to compress over twice as many blocks as previous algorithms, regardless of the quantization granularity. While practical algorithms will undoubtedly do worse, these bounds suggest that practical algorithms may do better than previously published algorithms.

5 BYTE-SELECT ALGORITHM SEARCH PROBLEM

This section describes our approach to searching for a practical byte-select compression algorithm. Practicality means both a good quantized compression ratio and a small, low-latency hardware implementation. We first show that a naïve byte-select algorithm results in simple hardware, but a poor quantized compression ratio. However, the insight gained from this initial failure suggests a more successful approach.

Fig. 4. Existing algorithms' performance is far from ideal.

Fig. 5. Metadata overhead of naïve byte-select algorithms increases with byte dictionary size, limiting compression.

5.1 Naïve Implementations Need Too Much Metadata

We first consider a naïve byte-select algorithm inspired by the idealized bound in the previous section. Like the idealized algorithm, our naïve algorithm only stores one copy of each unique byte in its dictionary, thus minimizing dictionary size. However, unlike the idealized algorithm, our naïve algorithm must encode the mapping from every byte in the byte dictionary to every byte in the uncompressed block. Our straightforward implementation requires $\lceil \log_2(n) \rceil$ metadata bits for each byte in the n -byte dictionary. Metadata size very quickly dominates directory size; an 8-byte dictionary requires $64 * \lceil \log_2(8) \rceil \text{ bits} = 24 \text{ bytes}$ of metadata.

Figure 5 shows the performance of the algorithms for different dictionary sizes. The maximum possible compression ratio is for a 4-byte dictionary (i.e., 2 bits of metadata required per byte). The unquantized compression ratio of 1.24 is much lower than both the upper bound of 3.4 (as seen in Figure 4) and the best compression ratio of an existing algorithm, 1.6.

Unlike the unquantized compression ratio, quantized compression goes to zero for dictionaries larger than 8 bytes since a 9-byte dictionary would require 32 bytes of metadata. The second important difference is that 32-byte quantized compression ratio is highest for the 8-byte dictionary algorithm, the maximum byte dictionary size possible. When we measure algorithms by quantized compression ratio, decreases in size below the target are no longer seen as useful.

5.2 Cache Block Patterns

To limit metadata size, we must restrict the possible mappings from the byte dictionary to the output bytes. Graphically, this corresponds to removing some of the arrows on the right side of Figure 1. Our initial approach was to estimate the frequency of each individual byte mapping (i.e., an individual arrow) and then search for frequent patterns. However, we did not find an efficient way to search this space. Instead, we focus on the simpler problem of identifying frequently used full cache-block mappings. Graphically, such a mapping corresponds to any subset of the rightmost arrows in Figure 1 such that exactly one arrow points to each output byte.

In this work, we represent these mappings as *cache block patterns*, each a string of symbols, one per byte in the cache block. A compression algorithm generated by our search corresponds to one or more of these patterns. Each symbol in a pattern can be interpreted as an index to a byte in the byte dictionary. Figure 6 illustrates how two cache block patterns can be combined with three different byte dictionaries per pattern to encode three different 4-byte cache blocks. The first pattern, 0101, only uses two bytes in the dictionary, while the second pattern, 0102, uses three. Note that different patterns can be used to describe the same cache block. We say that a cache block is *describable* by a pattern if bytes with the same symbol in the pattern have the same

Fig. 6. Patterns with three of their describable cache blocks. A few patterns can describe many cache blocks with variation in a byte dictionary.

Fig. 7. Example of Byte-Select (de)compressor using chosen patterns.

value in the cache block. Notice that in some cases, the dictionary has multiple entries with the same value. Thus, this approach re-introduces duplicates into the dictionary, decreasing effective dictionary size, in order to (greatly) reduce metadata size.

Assume our search has identified the two most useful patterns: 0112 and 0120. Figure 7 illustrates block diagrams for compression and decompression hardware for the resulting algorithms. The compression block must check, for each pattern, whether the bytes with the same symbol have the same value. In this example, comparators check the two middle bytes for the first pattern and the two outer bytes for the second. If all the comparisons required for a given pattern are successful, that pattern describes that cache block. A priority encoder selects one of the matching patterns (if any), writing the **pattern identifier (PID)** to the compressed output. A set of associated selectors decode the PID to map bytes from the input block to the byte dictionary in the compressed output. Note that the selector inputs are hardwired to select the correct input byte for each pattern. In this example, for pattern 0112, the byte dictionary will contain the bytes “abd”, while for pattern 0120, the byte dictionary will contain bytes “abc”. Thus, the compressor is simply a set of hardwired comparators, a priority encoder, and a set of hardwired multiplexors.

The corresponding decompressor, shown on the right, is even simpler. The PID determines the output selectors, which are hardwired to map bytes in the dictionary to the output bytes.

5.3 Pattern Search Problem

The remaining problem is to choose a set of patterns that meets the following conflicting goals. First, the set of patterns should be large enough to achieve a high quantized compression ratio. That is, the set of patterns must describe a large fraction of blocks in most workloads. Second, the set of patterns must be small enough to limit the metadata size. For p patterns, the compressed representation requires $\lceil \log_2(p) \rceil$ bits of metadata. Large numbers of patterns will tend to reduce the number of bytes available for the byte dictionary. In addition, larger numbers of patterns will increase the area and latency overheads for the compression and decompression logic. Fewer patterns mean fewer comparators and a smaller priority encoder in the compressor, plus fewer inputs to the multiplexors in the compressor and decompressor.

Finally, the selected patterns should be chosen based on the available compressed output sizes. As noted above, compressing to fewer bytes than a quantized output size provides no benefit unless it compresses to a smaller quantized size. Thus, our search algorithm will take as input a set of compressed output sizes and identify a set of patterns for each size. In this work, our search strategy does not consider interaction between patterns for different target sizes. Rather, for simplicity, we decompose the multi-target-size problem into separate search problems for each

Fig. 8. Algorithm generation flow.

size, using simple encoding techniques to differentiate the different sizes in the metadata. Thus, the core search problem becomes: from the set of all possible patterns, *pick the best m patterns with byte dictionary size no greater than n* . The set of patterns chosen maps to an implementable compression algorithm, i.e., some hardware that can compress any cache block describable by any of the patterns in the set.

6 ALGORITHM GENERATION

Figure 8 illustrates the high-level flow of our search algorithm to find the best cache block patterns. In order to rank patterns, we will first count how often they occur in training traces of cache blocks. We then use multiple steps (*phases*) to cut down the set of patterns based on different characteristics. Finally, we use the selected patterns to generate RTL.

6.1 Trace Processing

The first step in the generation process is to derive pattern counts from our training traces. A straightforward method would be to count how many cache blocks are describable by each pattern. However, there are a huge number of possible patterns and a cache block is describable by multiple patterns. Directly counting like this would both be computationally expensive and would lose some useful information about which patterns share which blocks. Instead, our approach is to compute the number of cache blocks for which each pattern is the simplest describing pattern (i.e., contains the fewest unique symbols) that can describe the cache block. An example of this is shown in Figure 9. To compute the simplest pattern for a cache block, we can first compute an array of unique bytes in the order they appear. Then each byte in the original block is replaced with its value's index in the array. It is easy to show that there is only one simplest describing pattern for any cache block.

We then summarize the trace by counting the number of occurrences of each pattern, c_i , for a given pattern with index i . A result of the one-to-one relationship between patterns and cache blocks is that each count represents a disjoint set of cache blocks. This makes them easier to work with when we search for patterns that describe multiple sets of cache blocks in future steps.

6.2 Pattern Set Reducer

Now we have a set of patterns with associated counts. We need to pick the best m patterns that require no more than n bytes in the byte dictionary. We will use a four-phase process that

Fig. 9. Cache blocks with their simplest describing patterns.

Fig. 10. Pattern Lattice for 4-byte Cache Blocks.

successively cuts more patterns in each phase. An overview of these phases is shown in the table in Figure 8. Each phase has equal or greater algorithmic complexity than the last, described in the rightmost column of the table where n_i is the number of patterns input to phase I and m is the final number of patterns. The first phase takes as input the patterns and counts described in the last subsection. The final phase outputs a sorted set of patterns from which the top p can be used to generate RTL.

6.2.1 Removing Infrequent Patterns. In the first phase, we remove any pattern that is the simplest describing pattern for less than a threshold number of cache blocks. The threshold is given as input to the algorithm generation process. The threshold value is chosen to allow the later phases to finish in an acceptable amount of time as defined by the user. A higher threshold means fewer patterns for the later, more complex, phases to process. There may also be some benefits in terms of regularization (i.e., we have seen a higher threshold improve compression results of later phases), but we do not study this in detail. Unlike later phases, we are not making any guarantees about the usefulness of the patterns we are cutting in this phase. We cut patterns in this phase based on the heuristic that patterns appearing less often should be less useful. The output from this phase is in the same format as that of the trace processing: patterns and counts, c_i , that indicate how many cache blocks were most simply described by a given pattern.

6.2.2 Removing Unnecessary Patterns. In the second phase, we remove redundant patterns. We observed earlier that multiple patterns can describe a given cache block. We can create a partial order of patterns based on the sets of cache blocks they describe. We say that one pattern is less than another pattern if the set of cache blocks the first describes is a subset of the cache blocks described by the second. This partial ordering allows us to create a *bounded lattice* [23] of all patterns. An example of one such lattice for 4-byte cache blocks is shown in Figure 10. Arrows denote that the source pattern is less than the destination pattern. Notice that the patterns in this lattice can be *ranked* according to the number of unique bytes that appear in the pattern. The bottom of the lattice is the pattern with one unique byte (on the left side of the figure), and the top is the pattern with the maximum number of unique bytes, four in this case (on the right side of the figure).

Since our search focuses on an n -byte dictionary, we are only interested in patterns with $rank \leq n$, i.e., n or fewer unique bytes. So, we first discard patterns from the trace with $rank > n$ that would require a directory larger than size n . We then look at which of the remaining patterns are *maximal*; that is, patterns that are not less than any other remaining pattern. Non-maximal patterns are redundant in that all the cache blocks they describe can also be described by maximal patterns. Since we have partitioned our search and are only considering a single target dictionary

size, patterns with fewer unique bytes provide no additional compression benefit. Therefore, these patterns are redundant in their ability to increase quantized compression ratio as well.

Consider the example lattice in Figure 10, and let $n = 3$. The shaded elements show that three patterns: 0000, 0010, and 0011 appeared in the trace. However, the pattern 0000 is non-maximal, as the other two patterns are both greater than it. The remaining two patterns are maximal, as there are no larger patterns that appeared in the trace.

While we can eliminate non-maximal patterns from the search, we do not want to eliminate their counts. In this stage we compute the total number of cache blocks described by each of the maximal patterns. A maximal pattern p_i 's count, $C_i = \sum_{p_j \leq p_i} c_j$, is simply the counts of all patterns p_j less than or equal to itself. Since a non-maximal pattern may be less than multiple maximal patterns (i.e., a given c_j may appear in multiple summations for different C_i 's), we must be careful not to double count in later stages. To do this, we maintain the set of non-maximal patterns and counts (but exclude them from the search itself). This guarantees that we continue to describe the same set of cache blocks.

6.2.3 Combining Maximal Patterns. Patterns may be maximal but have rank less than n . In this stage we seek to combine such patterns to form new maximal patterns with higher rank no more than n , thus describing the same set of cache blocks with fewer patterns. Given two patterns in our lattice, we can replace them with their *least upper bound* pattern if it has rank no more than n . The least upper bound of a pair of patterns, (p_i, p_j) , is defined as the lowest rank p_k such that $p_i \leq p_k$ and $p_j \leq p_k$. Like before, this new maximal pattern's count is the sum of all lesser patterns' c_i 's, so combining the two patterns does not result in double counting.

Figure 10 illustrates such a combining step for our constraint of $n = 3$. The two patterns that remained from the first phase, 0010 and 0011, have rank 2. Their least upper bound is the pattern 0012, with rank 3. By replacing these two patterns with the pattern 0012, we can describe all the same cache blocks as before while using fewer patterns.

We use a greedy heuristic to combine patterns in this phase. The patterns are sorted lexicographically, then iterated over with a nested loop to generate candidate pairs. If their least upper bound has rank no greater than n , then the two patterns in the pair are replaced with the new pattern. Any other maximal pattern that is less than the new pattern is also removed. This process iterates over the patterns, including the new replacement patterns, until no pairs are combined. All the remaining patterns are maximal, with rank no more than n , and collectively describe all the blocks that survived the initial, threshold screening.

6.2.4 Iteratively Picking Best Patterns. The final phase iteratively selects the best m patterns. Each iteration sorts the patterns by C_i . The pattern with the highest count, p_{best} , is selected (ties are broken arbitrarily) and removed. To ensure we get the best pattern on the next iteration, all cache blocks described by the selected pattern are removed. This is equivalent to removing from consideration any (non-maximal) pattern $p_i \leq p_{best}$ and its corresponding count c_i . We then recursively recompute C_i for any remaining pattern, thus guaranteeing that the next "best" pattern is not selected due to cache blocks that are already described. We use A* search [24] so we can skip the recomputation of some C_i 's while still getting the same result. Note that a guarantee that we are choosing the best pattern given the current sorted list is weaker than a guarantee that we are choosing the best set of patterns overall. However, our method results in a single sorted list of patterns that can be reused for any value of p smaller than the size of our sorted list.

6.3 Generate RTL

After the fourth phase is completed, we have a set of m patterns that respect our byte dictionary size constraint n . Now we generate RTL for the compression and decompression processes using

these patterns. The process we use is the same as was demonstrated in Figure 7. The compressor includes a success check for each pattern that determines if the cache block input is describable by that pattern. This requires comparing any bytes that have the same symbol in a pattern. The byte dictionaries for every pattern are created, and the final byte dictionary is chosen based on the first pattern that successfully compresses to the desired size. The index of the pattern is stored as metadata. The decompressor recreates the original cache block using the byte dictionary assuming each pattern was successful. Then the correct cache block is selected using the pattern index stored in metadata. The patterns are hard coded in the RTL, allowing synthesis to optimize the logic and greatly improve the performance and area characteristics of the hardware compared to a non-hardcoded implementation (Section 8.3).

7 METHODOLOGY

7.1 Algorithm Generation and Trace Based Evaluations

We trained using traces generated by various architectural simulators. We evaluated workloads for three sets of traces shown in previous figures. The CPU traces were generated using the SimNow simulator. SPEC CPU^{®1,2}2006 [25] as well as a set of HPC apps including LULESH [26] are in this trace set. The traces are generated by collecting data from backing stores in gem5. Whenever there is a miss in the LLC, a request is sent to the main memory via the memory controller. Then, the memory controller gets the cache line from the main memory to send the LLC. In this case, the trace generator records the content of the cache line as seen by the memory controller. The GPU Compute trace set includes LULESH and some applications from the Rodinia suite [27]. The GPU Game traces were generated with an in-house simulation infrastructure.

To build training sets using these traces, we set aside the middle 10% of each trace for testing and use the rest for training. Then we train with combinations of the traces corresponding to the sets described earlier, always using the same 90%. For the CPU traces, we use a pattern count threshold of 1024 in the first phase. For GPU traces, we use a threshold of 16. These thresholds were chosen after testing powers of 4 and choosing one that finishes training in under a day.

We present results for four different generated algorithms. The first two (BSel-256/4096) are trained on 90% of every trace and include 256 or 4096 of the top patterns. These algorithms require one and two metadata bytes respectively to describe the pattern identifiers. So, the two algorithms were trained separately with 31 and 30-byte dictionary size constraints respectively. The “BSel-1024-1024-128” algorithm includes 1024 patterns for 32 and 16-byte compressed outputs and 128 patterns for 8-byte compressed outputs. We limit the number of 8-byte patterns to 128 patterns so we can use a 7-byte dictionary for those patterns. In the metadata, we use one bit to signify whether the pattern is for 8 or 32/16-bytes, then 7 bits to index the 8-byte patterns and 15 bits to index the 32/16-byte patterns.

7.2 RTL Evaluations

We generate and synthesize RTL for a selection of the algorithms generated using those training sets. We wrote a reconfigurable version of the decompression logic by hand. This version stores patterns in registers but is otherwise the same as the generated algorithms. We synthesize all RTL with a 14 nm cell library using Synopsys Design Compiler. All (process technology-agnostic) area and latency numbers for existing algorithms are from their respective publications.

¹SPEC and SPEC CPU are registered trademarks of the Standard Performance Evaluation Corporation.

²See www.spec.org for more information about SPEC[®] benchmarks.

Table 2. Simulation System Architecture

CPU	1400MHz 2 core - Zen™ μ Arch
L1	32kB – 8 ways – private - Cache_Line_Size=64Bytes
L2	1MB – 8 ways – private - Cache_Line_Size=64Bytes
L3	16MB – 16 ways – shared - Cache_Line_Size=64Bytes
GPU	1700MHz 64 CUs
L1	16kB – 16 ways – 1 bank – private write-through – Cache_Line_Size=128Bytes
L2	256kB – 16 ways - 16 banks – shared write-back – Cache_Line_Size=128Bytes – L2_Read_Energy = 10.45pJ – L2_Write_Energy = 6.27pJ
Memory	HBM2_1000_X128 – 16GB capacity C/R/B=8/1/16 – Page size 2KB - Bus_Width = 128bits – tCK = 1ns tCK/tBURST/tCCD_L/tRCD/tCL/ tRP/tRAS/tRRD/tRRD_L/tXAW /tRFC/tWR/tWTR/tRTP/tRTW/tXP/tXS 1/16/4/12/6/14/33/5/6/20/350/16/8/5/5/7. 5/360 Mem_Access_Energy= 29246pJ

7.3 System Level Evaluations

To evaluate the effects of data compression on system performance, we evaluate BSEL-1024-1024-128 in gem5 on SPEC CPU®^{1,2} 2006 and GPU Compute and Game applications. Our simulated system models an AMD CPU and GPU each with a multi-level cache hierarchy (Table 2). Link compression is applied to the data travelling on the memory channel between LLC and DRAM in GPU evaluations, but not in CPU evaluations. The CPU and GPU systems include an implementation of YACC [12], which we received from the original author and modified to work with an in-house branch of gem5. Compression is applied to the LLC in both CPU and GPU evaluations. We removed some applications from each set which would not be expected to see improvement from compression, to isolate the effect of our new algorithms. For CPU results, we removed three applications which did not complete simulation properly (milc, libquantum, and dealII).

8 RESULTS

8.1 Compression Ratio

We present quantized compression ratio results for different existing algorithms as well as a selection of generated algorithms in Figure 11 and Figure 12. Among existing algorithms, there is a significant variation in performance across workload types. CPack and FPC were designed for CPU applications, where most of the data are integers and characters. Unsurprisingly, they do the best for CPU workloads among existing algorithms. BPC does best for GPU workloads which have more diverse data types, including integer, floating point, and pixel color data.

Our generated algorithms do not suffer from the same trace-type dependence that existing algorithms suffer from. Despite using byte boundaries, they can compress traces with varying data

Fig. 11. Performance of existing and generated algorithms for 32/16/8-byte quantized compression.

Fig. 12. Performance of existing and generated algorithms for 32-byte quantized compression.

Fig. 13. Performance when using patterns from target trace sets is not much higher than using patterns from all sets.

types as well as or better than BPC depending on the quantization type. They also do much better than CPack at compressing traces with mostly integer data. This leads to a 7% improvement in 32-byte quantized compression ratio when using BSel-4096 and 23% improvement in 32/16/8-byte quantized compression ratio when using BSel-1024-1024-128 on average compared to the best existing algorithm.

The algorithm trained with multiple target sizes does better relative to other algorithms for 32/16/8-byte quantization but worse for 32-byte quantization. The difference is especially apparent for GPU Compute applications where we are otherwise beat by BPC. This demonstrates one key benefit of an automated search process: we can automatically create algorithms targeting different hardware applications just by changing parameters of the search.

BSel-**-T* results are shown in Figure 13. The algorithms trained on target traces do not outperform the BSel-* counterparts by more than 3% in any case. This tells us that the set of useful patterns is either small enough or general enough that we can successfully create an algorithm that does well for all trace types. However, if the designer is sure that they will only be targeting one type of data, that 3% may be worth the loss in generality and is a benefit of our automated process.

8.1.1 Useful Patterns. To try to understand why the generated algorithms are doing better than existing ones, we analyze the top two patterns for each trace set, shown in Table 3. Each number represents which byte in the dictionary should map to that corresponding byte in the output. For example, the first 8 bytes in the first pattern, 11112345, mean that the first four bytes in the output are the same and the next four bytes are all (potentially) different.

All the patterns are some variation of previously exploited power-of-two word size with deltas in the most significant or least significant bytes. Consider the first GPU Game pattern for example. The first two 4-byte words are 1234 and 1256. This means the words share the two most significant bytes in common and can differ in the lower two.

In a few of the CPU patterns, we see stretches of bytes described with the same code instead of unique bytes. Consider the first HPC pattern where some 4-byte stretches are 5555. This is likely describing adjacent zero bytes in those positions. This and the power-of-two word patterns are commonly exploited in existing algorithms. They are useful for any data type that has power-of-two boundaries which includes floating point and integer values (more floating point analysis is provided in Section 9.2). We hypothesize that the additional compression benefits come from the extension of these known patterns in small and inhomogeneous ways, changes that are less likely to be made by a human designer who values symmetry. Some of these differences can be described in intuitive ways.

The first explainable difference is that our patterns exploit locality within groups of adjacent words in the cache block. Many of the patterns are doing this with 4-byte words. Consider the first GPU Game pattern again for example. It is describing 4-byte words with 2-byte deltas in the least significant bytes. If we give unique deltas to every word however, we do not hit our 31-byte dictionary size constraint. To meet this constraint, the pattern allows the second least significant byte of cache-block adjacent words to match. For example, bytes 24-27 and bytes 28-31 share the third byte value in addition to the first two. We see a similarly describable effect in the more significant bytes of the second GPU Game pattern. For example, bytes 32 and 48 are at the same location in their respective 4-byte words and are allowed to take a shared value that is unique relative to that position in the rest of the 4-byte words. These patterns are exploiting the fact that, even within pieces of a cache block, there is spatial value locality. We have not seen spatial value locality exploited hierarchically in this fashion within cache blocks in any existing algorithm.

The second difference (observable in the previous examples as well) is that our patterns are not entirely regular. While a human designer may be put off by the lack of symmetry, the best patterns will exploit symmetrical structure in the data while utilizing the whole byte dictionary. Consider the first GPU Compute Pattern for example. It is targeting cache blocks that have one 8-byte word repeated across the block. A human designer may, observing this pattern, just repeat symbols 1 through 8 across the whole cache block (this exact pattern is in BDI). That pattern would be symmetrical, but it would not use the full 31-byte dictionary available. Instead, the generated pattern allows the two 8-byte words starting at bytes 48 and 56 to be completely unique, increasing the set of cache blocks compressed by the pattern.

Both examples we presented take obvious patterns and make small changes that are not immediately obvious. While small changes like these can be useful, it is not always obvious which one is the most useful since they do not fit the more obvious and symmetric pattern. A tool for learning patterns is especially useful in this scenario since it can easily quantify benefits in a way that would be labor intensive for a human. We submit that the main benefit of our generator, just like any machine learning tool, is its ability to make intricate, non-obvious changes to the existing state-of-the-art. Fortunately, in many cases like those above, the changes make intuitive sense.

Table 3. Top Patterns From Different Trace Sets

	SPEC2K6		HPC		GPU Compute		GPU Game	
0	1	1	1	1	1	1	1	1
1	1	2	2	2	2	2	2	2
2	1	3	3	3	3	3	3	3
3	1	4	4	4	4	4	4	4
4	2	5	5	1	5	1	1	5
5	3	6	5	2	6	2	2	6
6	4	7	5	3	7	5	5	7
7	5	8	5	5	8	6	6	8
8	1	9	6	1	1	1	1	9
9	1	10	7	2	2	2	2	10
10	1	3	8	3	3	7	7	3
11	1	16	9	6	4	8	8	11
12	2	17	5	1	5	1	1	5
13	3	18	5	2	6	2	2	12
14	6	19	5	3	7	7	9	7
15	7	20	5	7	8	9	10	13
16	1	9	28	1	1	1	1	1
17	1	10	10	8	2	2	2	14
18	1	3	11	9	3	10	11	3
19	1	11	12	10	4	11	12	15
20	2	12	5	1	5	1	1	5
21	3	13	5	8	6	2	2	16
22	8	14	5	9	7	12	13	7
23	9	15	5	23	8	13	14	17
24	1	9	29	1	1	1	1	9
25	1	10	13	8	2	2	2	18
26	1	3	14	24	3	14	15	3
27	1	11	15	12	4	15	16	19
28	10	12	5	1	5	1	1	5
29	11	13	5	8	6	2	2	20
30	12	14	5	25	7	16	15	7
31	22	15	5	26	8	17	17	21
32	1	21	6	1	1	1	1	22
33	1	22	16	8	2	2	2	23
34	1	3	17	27	3	18	18	3
35	1	23	18	14	4	19	19	24
36	23	24	5	1	5	1	1	5
37	24	25	5	8	6	2	2	25
38	14	26	5	28	7	18	20	7
39	15	27	5	15	8	20	21	26
40	1	9	1	1	1	1	1	27
41	1	10	19	8	2	2	2	28
42	1	3	20	29	3	21	22	3
43	1	11	21	16	4	22	23	24

(Continued)

Table 3. Continued

	SPEC2K6		HPC		GPU Compute		GPU Game	
44	25	12	5	1	5	1	1	5
45	26	13	5	8	6	2	2	25
46	16	14	5	30	7	23	24	7
47	17	15	5	17	8	24	25	26
48	1	9	30	1	9	1	1	22
49	1	10	22	8	10	2	2	29
50	1	3	23	31	11	25	26	3
51	1	11	24	18	12	26	27	19
52	27	12	5	1	13	1	1	5
53	28	13	5	8	14	2	2	30
54	18	14	5	19	15	27	26	7
55	19	15	5	20	16	28	28	21
56	1	9	31	1	17	1	1	27
57	1	10	25	8	18	2	2	31
58	1	3	26	19	19	29	29	3
59	1	11	27	21	20	30	30	19
60	29	12	5	1	21	1	1	5
61	30	13	5	8	22	2	2	30
62	31	14	5	19	23	29	29	7
63	21	15	5	22	24	31	31	21

Table 4. Baseline (No Compression) Measurements

CPU MPKI				GPU BW (GB/s)	
401.bzip2	3.894614	<u>473.astar</u>	1.482074	MiniFE	103.1125
437.leslie3d	8.614004	403.gcc	4.080995	HPGMG	14.04727
444.namd	0.087038	433.milc	19.47066	<u>Quicksilver</u>	35.09618
435.gromacs	0.835193	<u>436.cactus</u>	5.379156	CoMD	64.01903
459.GemsFDTD	8.141758	453.povray	0.007605	PENNANT	88.83998
471.omnetpp	20.36931	447.dealII	26.94446	XSbench	112.7284
429.mcf	75.3734	450.soplex	28.81943	HACC	0.800323
470.lbm	30.23084			<u>SNAP</u>	3.513147
458.sjeng	0.365222			<u>LULESH</u>	15.23115
462.libquantum	23.57521			bfs	114.7902
483.xalancbmk	2.368641			spmv	2.072129
454.calculix	0.017583			<u>hotspot</u>	98.1656

8.2 System Level Evaluations

Table 4 shows MPKI and read bandwidth measurements for the CPU and GPU applications. Applications that saw at least a 10% improvement in their respective metric from any compression algorithm while seeing >1 MPKI in the case of CPU applications or consuming >1% of available link bandwidth in the case of GPU applications are underlined.

We used BSEL-1024-1024-128 in all system level evaluations. Figure 14 shows the improvement in MPKI when using an implementation of YACC with various compression algorithms. We see

Fig. 14. MPKI for SPEC2K6 applications.

Fig. 15. Read bandwidth for GPU Compute applications.

that in all cases, the improvement from BSel is close to or exceeds the best performing algorithm. For some applications we observed an increase in MPKI. The only application for which the initial MPKI was significant and also increased was bzip. The increase for other applications can be explained by noise and a low initial MPKI. For bzip, the compression ratios we saw were no greater than 1.1 for any algorithm. This makes sense given that much of the data should be already compressed. As a result of the low compression ratio, the additional latency from compression results in an increase in MPKI.

Figure 15 shows the reduction in read bandwidth from compression for GPU workloads. We see again that whether CPack or BPC has the highest improvement, the improvement from BSel-1024-1024-128 is similar or greater in all cases except bfs. The bandwidth improvement from Byte-Select ranges from 2%–36%, demonstrating the high variation in compressibility for these workloads. One explanation for this variability could be an uneven distribution of hard to compress double precision floating point values.

We show speedups for CPU and GPU applications in Figure 16. In most cases, the MPKI and bandwidth improvements translated to proportional speedups. Some exceptions in GPU applications like hotspot exist. This is because increasing effective available bandwidth using compression

Fig. 16. IPC improvement for all applications.

only improves performance up to the point where bandwidth is no longer a bottleneck. So, we see BDI and CPack have no bandwidth improvement while having a substantial speedup. BPC and BSeI, on the other hand, see both a speedup and a bandwidth reduction since they compress more and reduce bandwidth past the point where it is a performance bottleneck.

8.3 RTL Evaluations

In addition to good compression ratios, we need low hardware costs in terms of area and latency. We show the latency and area (compressor+decompressor) of the algorithms we presented as well as existing algorithms in Table 5. We also show the average quantized compression ratio for quantization to 32, 16, and 8 bytes as well as only 32 bytes. Our algorithms show significant improvement in latency compared to any of the previously top performing algorithms. With our 256-pattern algorithm, both compression and decompression can be completed in a single cycle, while achieving better than state-of-the-art compression ratio for the 32-byte quantized compression ratio and

Table 5. Hardware Characteristics of Cmpr. Algorithms

	Area (gates)	Latency (cycles)		CR-32/16/8	CR-32
		C	D		
BSel-256	64k	1	1	1.35	1.27
BSel-4096	440k	2	1	1.56	1.30
CPack	26k	13	8	1.44	1.23
BPC	68k	7	7	1.48	1.20
BSI	9k	1	1	1.38	1.16
FPC	-	3	5	1.44	1.14

a similar 32/16/8-byte quantized compression ratio to the only algorithm with similar latency, BSI.

Moving to 4096 patterns increases compression latency to two cycles. The area of our 256-pattern algorithm is comparable to BPC. The area of the 4096-pattern algorithm is significantly higher than that, but still very small. For example, in a processor made up of a billion gates, 440k gates is only 0.044% of the total area. The area and latency of BSel-1024-1024-128 should be somewhere between BSel-256 and BSel-4096 since it has about 2 thousand patterns total, and all patterns use 30 bytes or less in their byte dictionary.

Since we are generating a single algorithm for CPU and GPU traces, there are also hard-to-quantify benefits in terms of hardware design costs. Optimizations to the generated RTL can be performed one time and then the RTL module can be used as a black box IP across designs. The design can also be used in shared graphics memory systems where both CPU and GPU data can reside.

We also wrote a reconfigurable version of the pattern-based decompressor. We found that, for our naïve representation of patterns, the area overhead was unacceptable. About 705 thousand gates are required for a 64-pattern decompressor. We also looked at the area for a single reconfigurable pattern to get an upper bound on the area of the byte muxes. With a single reconfigurable pattern, there will be a mux for every byte in the output with all possible byte dictionary bytes as input. The area of the single pattern decompressor is 15 thousand gates, significantly lower than other algorithms, and timing constraints were met. This suggests that byte muxes are not greatly impacting complexity of the hardware.

The reconfigurable area required for 64 patterns is almost two times the area required for 4096 hard coded patterns. This suggests that there is a significant amount of pattern specific optimization being done by the synthesis tool. While the RTL specifies 64 4096-input multiplexers, the design tool can recognize and combine shared logic (e.g., the same byte from the dictionary being used in different multiplexers). This results in much lower latency and area than one may expect from a naïve hardware implementation of a 4096-pattern compressor. Such a naïve implementation is forced in the reconfigurable design because the patterns are not known at synthesis time.

8.4 Impact of Thresholds

As discussed in Section 6, the first phase of the algorithm takes as input a threshold on the number of times a cache data block pattern must appear for it to be considered in the later phases, a pattern count limit. This threshold is used to limit the execution time of the algorithm search. It also takes as input a target byte dictionary size; any pattern that requires more than that many unique bytes is discarded. Figure 17 shows the percentage of patterns remaining at the end of each phase relative to the beginning of the phase for different pattern count limits, input training traces, and target byte dictionary sizes. Missing data points are due to those searches breaking a running time constraint of approximately 24 hours. The input to phase 1 when considering all patterns (left side of the

Fig. 17. Percentages of patterns remaining after each phase.

figure) is about 400 million patterns while there are only 4 million in the GPU Game traces (right side of the figure). To achieve a similar number of patterns input to later phases for all training sets, we used a higher pattern count limit when considering all traces (1024) versus only GPU Game traces (16) to generate the results shown in previous subsections. We also used a pattern count limit of 1024 for CPU traces and 16 for GPU Compute traces since they started with a similar number of patterns as the collection of all traces and GPU Game traces respectively.

More than 99.9% of patterns are removed in the first phase when training on all traces since they occur fewer than 1024 times. We also see that >90% of patterns appear four or fewer times even for GPU Game traces. These results imply the majority of patterns will be of limited use, and it is ok to discard them. Both larger pattern count limits and smaller target byte dictionary sizes decrease the set of patterns we are interested in, and therefore result in a higher percentage of patterns from the traces being removed. The percentage of patterns identified as maximal by the second phase also decreases with the target byte dictionary size, possibly because the percentage of all acceptable patterns (agnostic of any trace) that are maximal decreases with an increasing

byte dictionary constraint. Relative to the first phase, both the second and third phases are not removing many patterns.

9 DISCUSSION

9.1 Improving the Algorithm Search

Some improvements should be possible to our automated generation process as well. In Section 8.3, we showed that the naïve implementation of a reconfigurable byte-select decompressor consumes too much area. Most of this area is consumed by multiplexors that output all the possible uncompressed cache blocks, one possible block for each pattern. When the patterns are hard-coded, these multiplexors are unnecessary and optimizations across the patterns are possible. It may be possible to analyze what cross-pattern optimizations are being done by the synthesis tool and then try to generalize and backport them as constraints on the reconfigurable patterns. It is also not obvious what the right way to store patterns is and what the costs involved in reconfiguration are and how they impact the design.

A better search process could also improve the algorithms generated. For our specified search problem, we used various heuristics for which there may be better alternatives. We about halved the difference between existing algorithms and the upper bound quantized compression ratio for 32-byte quantization, so there may still be significant room for improvement. There are also alternate versions of the search problem that could lead to better compression. Changes to the problem itself could also improve the algorithms. For example, the search process may change when considering all target quantized sizes at once. Different pattern symbol boundaries other than at the byte-level could also impact results.

9.2 Comparing Byte-Select Upper Bounds to Alternatives

We would like to be able to compare unique byte-based ideals to long standing information theoretic ideals [28]. The information theoretic ideal algorithm stores each byte in a number of bits equal to its Shannon entropy, which depends on the probability of the byte's value. While we do not have perfect knowledge of a distribution from which byte values are sampled, we will use the zeroth-order empirical entropy to get an estimate. The zeroth-order empirical entropy calculation assumes that all samples are independent, i.e., assume all individual byte values are sampled from a single unchanging distribution independent of each other. Under this assumption, the probability of each value is the number of times it appears divided by the total number of bytes.

We compute the empirical entropy for each cache block in a trace, and average over all the cache blocks to obtain results for the final algorithm in Figure 18. We see a mean compression ratio of 3.9, close to the compression ratio for the unique bytes without metadata measurement. This ideal assumes that each cache line has its own encoding for byte values without having to store that encoding anywhere, like our assumption that we can store only unique bytes without metadata. The entropy measure is equivalent to the compressed data size achievable by a Huffman code if the codebook storage was free. In a practical implementation of such a scheme, the codebook is potentially unique for each cache block and so would need to be stored with the compressed data.

Previous work has shown that it can be beneficial to be aware of the data types being compressed. This awareness can be used to design a compression algorithm that uses value boundaries other than byte boundaries [8, 21, 29]. Figure 19 shows the upper bound on the compression ratio of algorithms using byte boundaries, single precision floating point boundaries, and double precision boundaries. We see that for all application sets, byte boundaries perform the best. This

Fig. 18. Upper bounds on compression ratios for byte-select algorithms under different assumptions. Assuming one byte of metadata does not hurt compression ratio, but quantizing compressed sizes does.

Fig. 19. Performance of ideals based on byte boundaries versus floating point.

suggests that, unless we want to support multiple boundaries in hardware, we should concern ourselves with byte boundaries. We limit ourselves to byte boundaries in this work.

Even when data is in the floating point (or any other byte unaligned) format, byte-level patterns can still be used to compress the data. For example, single precision floating point values that are similar may only differ in the least significant bits of the mantissa. These differences can be captured in exactly the same way as differing least significant bits of an integer: by allowing differing values for the least significant byte in the 4-byte word. The practical utility of this ability of byte-level patterns is demonstrated by the fact that our generated algorithm beats BPC, an algorithm exploiting bit-level patterns.

9.3 Representing Prior Works using Byte-Select

Our final consideration will be how previous works can be described as byte-select compression algorithms.

FPC: Frequent Pattern Compression (FPC) algorithm treats the uncompressed input block as a collection of independent 4-byte words. It represents each word in the compressed representation using 3 bits of metadata and 0-4 bytes of data. FPC relies on significance encoding based on the insight that typical values are often much smaller than the maximum representable value. Because of this, many words can be represented sign-extending 0, 1, or 2 of the least significant bytes. The implementation of FPC is essentially byte selection, where each output byte selects between an entry in the byte dictionary and the fixed bytes 0x00 and 0xFF. Note that because FPC treats each word independently, entries in the byte dictionary are not shared by different words in the output.

CPack: Like FPC, CPack also treats the uncompressed input as a collection of 4-byte words and replaces each word with 2-8 bits of metadata and 0-4 bytes of data. Unlike FPC, CPack allows words in the output to share bytes in the byte dictionary. To reduce metadata overheads, CPack restricts this sharing to 4-byte aligned groups of bytes. Figure 20 shows a CPack decompressor for the 8-byte block example. The stored bytes are shown at the top left of the figure and are inputs to the muxes at the bottom during decompression. CPack's encoding bits are the metadata that is used to compute the mux selectors. The separated '00' mux input is an implicit zero value that the CPack algorithm allows as input to every mux. The latency of the CPack decompressor is dominated by the byte muxing. The maximum depth for a CPack compressor that takes 64-byte blocks as input is shown in Table 6.

Fig. 20. Byte selecting based implementation of various decompression algorithms.

MPack: The MPack compression algorithm is similar to CPack but seeks to reduce the decompression latency by only storing 4-byte words in 0, 2, or 4 bytes. This limits the maximum number of inputs to a mux in the decompressor. The maximum number of mux inputs for MPack is four, one less than CPack’s five (Table 6). The tradeoff for this lower latency is a lower compression ratio in some cases. Consider the running example. CPack only needed to store the least significant byte of the second word, 0×18 , while MPack needs to store the least significant halfword, $0x0018$.

BDI: The BDI compression scheme uses addition to combine an 8, 4, or 2-byte base with a 2 or 1-byte signed delta, which means it is not a byte-select compression algorithm. Both BDI and byte-select algorithms exploit spatial value locality (i.e., the observation that values tend to be close together), but BDI has the potential to do better because it handles the case where a base+delta addition generates a carry out that propagates into the more significant bits. In practice, these carry out cases are relatively infrequent. This ability also comes at the expense of storing multiple very different values. With some small modifications that only improve the algorithm (Section 3.2) a BDI-like algorithm fits in the byte-select class.

Table 7 shows which previous works on cache compression we can describe as byte selection. The only algorithm considered in our study that we are unable to describe as byte selection is SC2, which uses a Huffman coding computed at the beginning of a benchmark run. As discussed in

Table 6. The Complexity of (de)Compression Logic is Determined by the Complexity of the Metadata Computation as Well as the Byte Muxes

Algorithm	Average Number of Inputs to		Maximum Number of Inputs to	
	Mux	Lookup	Mux	Lookup
CPack	16.375	3	20	4
MPack	13	4	16	4
FPC	~18	3	22	3
BDIZero	4.125	5	6	5

The Complexity of Many Existing Decompressors is Dominated by the byte Muxes.

Table 7. Most Cache Block Compression Algorithms can be Represented as Byte-Select Algorithms

Algorithm	Representable/Why
BDI BSI	☒ Addition ☒
CPack	☒
FPC	☒
ZCA	☒
SC2	☒ Requires dictionary shared in time

Section 2, we do not consider cross-block compression in this work but believe the simplicity of byte selection will also be useful for cross-block compression.

10 CONCLUSION

In this article, we introduced a class of byte-select cache-block compression algorithms and showed that algorithms in this class have the potential to achieve compression ratios up to 2X greater than those of state-of-the-art algorithms. We devised a search strategy to find algorithms that not only improves compression ratio by 23%, but also has a single cycle decompression latency, which is much lower than competitive algorithms. We attribute the success of our search strategy mainly to two observations: (1) the algorithm design process should include awareness of quantization of compressed sizes in hardware and (2) limiting algorithms to byte-selection fosters efficient hardware without severely limiting compression performance. Since our search is automated and parameterized, we generate algorithms specifically tailored to different hardware applications.

ACKNOWLEDGMENTS

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

- [1] A. Arelakis and P. Stenstrom. 2014. SC2: A statistical compression cache scheme. In *ACM SIGARCH Computer Architecture News* 42 (2014), 145–156, Accessed: Jun. 30, 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2665696>.
- [2] A. R. Alameldeen and D. A. Wood. 2004. Adaptive cache compression for high-performance processors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on* (2004), 212–223, Accessed: Jul. 02, 2017. [Online]. Available: <http://ieeexplore.ieee.org/abstract/document/1310776/>.
- [3] Xi Chen, Lei Yang, R. P. Dick, Li Shang, and H. Lekatsas. 2010. C-Pack: A high-performance microprocessor cache compression algorithm. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* 18, 8 (2010), 1196–1208, Aug. 2010, DOI: [10.1109/TVLSI.2009.2020989](https://doi.org/10.1109/TVLSI.2009.2020989)
- [4] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (2012), 377–388.
- [5] S. Sardashti and D. A. Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013) 62–73.
- [6] B. Panda and A. Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), 1–12, Accessed: Jun. 30, 2017. [Online]. Available: <http://ieeexplore.ieee.org/abstract/document/7783704/>.

- [7] A. R. Alameldeen and D. A. Wood. 2004. Frequent pattern compression: A significance-based compression scheme for L2 caches. *Dept Comp Scie Univ Wis.-Madison Tech Rep* 1500, Accessed: Jul. 02, 2017. [Online]. Available: <http://ftp.cs.wisc.edu/pub/techreports/2004/TR1500.pdf>.
- [8] A. Arakelis, F. Dahlgren, and P. Stenstrom. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods (date?), 38–49, DOI: [10.1145/2830772.2830823](https://doi.org/10.1145/2830772.2830823)
- [9] T. M. Nguyen and D. Wentzlaff. 2015. MORC: A manycore-oriented compressed cache. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2015), 76–88, DOI: [10.1145/2830772.2830828](https://doi.org/10.1145/2830772.2830828)
- [10] G. Pekhimenko. 2016. Practical data compression for modern memory hierarchies. *Carnegie Mellon University* (2016)
- [11] S. Sardashti, A. Seznec, and D. A. Wood. 2014. Skewed compressed caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (date)*, 331–342.
- [12] S. Sardashti, A. Seznec, and D. A. Wood. 2016. Yet another compressed cache: a low-cost yet effective compressed cache. *ACM Trans Arch. Code Optim* 13, 3 (2016), 27:1–27:25, Sep. 2016, DOI: [10.1145/2976740](https://doi.org/10.1145/2976740)
- [13] J. Dusser, T. Piquet, and A. Seznec. 2009. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing* 46–55, Accessed: Jul. 02, 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1542288>.
- [14] V. Young, S. Kariyappa, and M. K. Qureshi. 2019. Enabling Transparent Memory-Compression for Commodity Memory Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 570–581. DOI: [10.1109/HPCA.2019.00010](https://doi.org/10.1109/HPCA.2019.00010)
- [15] B. Abali et al. 2001. Memory expansion technology (MXT): Software support and performance. *IBM J. Res. Dev* 2001.
- [16] A. Shafiee, M. Taassori, R. Balasubramonian, and A. K. Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *High Performance Computer Architecture (HPCA)*, 2014 IEEE 20th International Symposium on 638–649, Accessed: May 22, 2016. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6835972.
- [17] S. Kim, S. Lee, T. Kim, and J. Huh. 2017. Transparent dual memory architecture. *Presented at the PACT'17*, Accessed: Oct. 07, 2017. [Online]. Available: http://calab.kaist.ac.kr:8080/~jhuh/papers/kim_pact17.pdf.
- [18] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, and S. W. Keckler. 2017. Compressing DMA engine: leveraging activation sparsity for training deep neural networks. *ArXiv170501626 Cs*, May 2017, [Online]. Available: <http://arxiv.org/abs/1705.01626>.
- [19] L. Yang, H. Lekatsas, and R. P. Dick. 2006. High-performance operating system controlled memory compression. In *Proceedings of the 43rd annual Design Automation Conference* 701–704, Accessed: Jul. 02, 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1147086>.
- [20] V. Sathish, M. J. Schulte, and N. S. Kim. 2012. Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* 325–334, Accessed: Jun. 30, 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2370864>.
- [21] J. Kim, M. Sullivan, E. Choukse, and M. Erez. 2016. Bit-plane compression: transforming data for better compression in many-core architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture*, Piscataway, NJ, USA, 2016, 329–340, DOI: [10.1109/ISCA.2016.37](https://doi.org/10.1109/ISCA.2016.37)
- [22] A. Ghasemazar, P. Nair, and M. Lis. 2020. Thesaurus: Efficient cache compression via dynamic clustering. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2020, 527–540, DOI: [10.1145/3373376.3378518](https://doi.org/10.1145/3373376.3378518)
- [23] Lattice (order). 2019. *Wikipedia*. Oct. 25, 2019, Accessed: Nov. 23, 2019. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Lattice_\(order\)&oldid=922960068](https://en.wikipedia.org/w/index.php?title=Lattice_(order)&oldid=922960068).
- [24] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*, Third Edition.
- [25] J. L. Henning. 2006. SPEC CPU2006 Benchmark descriptions. *SIGARCH Comput Arch. News* 34, 4 (2006), 1–17, DOI: [10.1145/1186736.1186737](https://doi.org/10.1145/1186736.1186737)
- [26] I. Karlin. 2012. LULESH programming model and performance ports overview. LLNL-TR-608824, 1059462, Dec. 2012. DOI: [10.2172/1059462](https://doi.org/10.2172/1059462)
- [27] S. Che et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, Oct. 2009, 44–54, DOI: [10.1109/IISWC.2009.5306797](https://doi.org/10.1109/IISWC.2009.5306797)
- [28] C. E. Shannon. A mathematical theory of communication. *Bell Labs Tech. J.* 27, 3, 55.
- [29] P.-A. Tsai and D. Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, 229–242. DOI: [10.1145/3297858.3304006](https://doi.org/10.1145/3297858.3304006)

Received September 2020; revised February 2021; accepted April 2021