Practical Speculative Parallelization of Variable-Length Decompression Algorithms

Hakbeom Jang Channoh Kim Jae W. Lee

Sungkyunkwan University
Suwon, Korea
{hakbeom, channoh, jaewlee}@skku.edu

Abstract

Variable-length coding is widely used for efficient data compression. Typically, the compressor splits the original data into blocks and compresses each block with variable-length codes, hence producing variable-length compressed blocks. Although the compressor can easily exploit ample block-level parallelism, it is much more difficult to extract such coarse-grain parallelism from the decompressor because a block boundary cannot be located until decompression of the previous block is completed. This paper presents novel algorithms to efficiently predict block boundaries and a runtime system that enables efficient block-level parallel decompression, called *SDM*. The SDM execution model features speculative pipelining with three stages: **S**canner, **D**ecompressor, and **M**erger. The scanner stage employs a high-confidence prediction algorithm that finds compressed block boundaries without fully decompressing individual blocks. This information is communicated to the parallel decompressor stage in which multiple blocks are decompressed in parallel. The decompressed blocks are merged in order by the merger stage to produce the final output. The SDM runtime is specialized to execute this pipeline correctly and efficiently on resource-constrained embedded platforms. With SDM we effectively parallelize three production-grade variable-length *decompression* algorithms—zlib, bzip2, and H.264—with maximum speedups of 2.50× and 8.53× (and geometric mean speedups of 1.96× and 4.04×) on 4-core and 36-core embedded platforms, respectively.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel programming; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Performance

Keywords Parallelization, runtime, speculation, compression, multicores, embedded systems

LCTES'13, June 20–21, 2013, Seattle, Washington, USA.

1. Introduction

Compression algorithms are widely used for efficient communication and storage of data. Their application domains include image and video processing, audio processing, networking, and data backups. Regardless of the target domain, *variable-length coding* is commonly employed to maximize compression ratio. Typically, variable-length compression algorithms split the original, uncompressed data into fixed- or variable-length *blocks* and compress each block using variable-length codes, hence producing variable-length compressed blocks.

With adoption of multicores in all scales of system, significant effort has been made to parallelize and accelerate these algorithms. For example, both gzip [2] and bzip2 [1], two popular general-purpose compression utilities, have parallelized versions such as pigz [8] and pbzip2 [7]. Both pigz and pbzip2 demonstrate near-linear speedups to core count when *compressing* a file, but execute sequentially when *decompressing* it.

Although compression (encoding) generally takes more computation than decompression (decoding), [1] parallelizing the decompressor is very important since it often constitutes the critical path of a user application, hence making a perceivable difference in response time. This is particularly true for embedded platforms which cannot afford high-frequency cores due to cost and power constraints. Unfortunately, the decompressor is much harder to parallelize than the compressor because a block boundary cannot be located until decompression of the previous block is completed. Therefore, the main challenge in parallelizing variable-length decompression algorithms is to efficiently identify the boundaries of compressed blocks that can be independently decompressed.

Some parallel decompressors address this challenge by specializing compressors. Examples include insertion of padding bits at the end of each compressed block to effectively equalize the block length [8] and inclusion of hints from which block boundaries can be readily calculated [18]. Obviously, such decompressors have limited applicability only to compressed input streams that were created by the modified compressors. Overcoming this limitation, there are other decompressors that do not require any compressor-side support. For example, Klein and Wiseman exploit the *self-synchronizing* property of Huffman coding [19] to identify the boundaries of compressed blocks. However, this method can only be applied to *static Huffman coding*, which uses a fixed Huffman table, but not to more popular *dynamic Huffman coding*. Also, block boundary locations have nearly zero correlations through an

[1] Throughout this paper "compress" and "encode" are used interchangeably. So are "decompress" and "decode".

Algorithm	Domain	Lossless?	Compressor Parallelized?	Decompressor Parallelized?	Block Delimiter Pattern Exists?	Remarks
zlib (gzip) [15]	Data	✓	✓	×	×	Huffman coding
bzip2 [1]	Data	✓	✓	×	✓(48 bits)	Huffman coding
H.264 [3]	Video	×	✓	✓	✓(40 bits)	Arithmetic coding (CABAC)
jpeg [4]	Image	×	×	×	✓(16 bits)	Huffman coding
png [9]	Image	✓	✓	×	×	Huffman coding (zlib+wrapper)
Vorbis [13]	Audio	×	×	×	✓(24 bits)	Huffman coding (+vector quantization)

Table 1: Survey of variable-length compression algorithms

input or across different inputs to make it difficult to apply existing, domain-unaware value prediction algorithms [21, 23, 26, 28, 32].

This paper introduces novel high-confidence prediction algorithms to identify the block boundaries efficiently, and also proposes the SDM runtime system to effectively decompress multiple blocks in parallel using these algorithms. The execution model of SDM features speculative pipelining with three stages: Scanner, Decompressor and Merger. The scanner stage executes a prediction algorithm that finds block boundaries by *partial decompression* and *pattern matching*. This algorithm splits the input stream into multiple, well-aligned chunks of blocks that can be independently decompressed while taking only a fraction of total execution time. The parallel decompressor stage decompresses multiple chunks concurrently and passes the decompressed output to the merger, which in turn constructs the final output stream. To ensure the correctness of the final outcome, the SDM runtime implements misspeculation detection and recovery mechanisms, which are invoked when the prediction algorithm incorrectly predicts the starting point of a chunk. To make best use of a handful of cores in resource-constrained embedded platforms, SDM supports *distributed commit*, where merging is done by decompressor processes in a distributed manner.

Without modifying the compressor, we successfully parallelize three production-grade variable-length decompression algorithms using SDM: zlib (gzip) [15], bzip2 [1] and H.264 [3]. To demonstrate the practicality of SDM, we use two resource-constrained embedded platforms for evaluation, Samsung Exynos 4412 quad-core platform based on ARM's Cortex-A9 [10] (representing "fat" cores) and Tilera's 36-core platform [12] (representing "thin" cores). Compared with original program compiled with `gcc -O3`, the parallelized program achieves maximum speedups of 2.50× and 8.53× and geometric mean speedups of 1.96× and 4.04× on the ARM and Tilera platforms, respectively. Evaluation on these two platforms reveals interesting tradeoffs between the two commit modes that SDM supports.

In summary, this paper makes the following contributions:

- Introduction of efficient prediction algorithms to identify block boundaries for three widely-deployed variable-length decompression algorithms
- Design and implementation of the SDM runtime system to enable efficient speculative execution of parallel decompression on resource-constrained embedded platforms
- Detailed evaluation of the three variable-length decompression algorithms on ARM- and Tilera-based embedded platforms

2. Motivation

2.1 Variable-Length Decompression Algorithms

A compression algorithm adopting a variable-length coding (called *variable-length compression algorithm* for brevity) compresses input data by exploiting the probability distribution of the frequencies of input symbols. The most common symbol are usually assigned the shortest code word to maximize compression ratio. To adapt to phase behaviors of the input stream and improve error resilience,

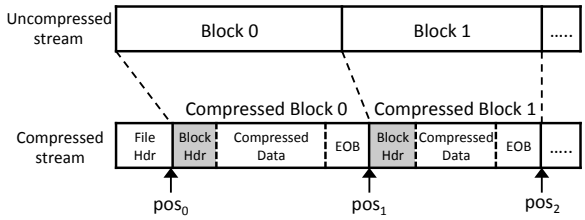


Figure 1: Variable-length compressed blocks created by compression algorithm (EOB stands for end-of-block and pos_n indicates the starting position of compressed block n)

the compressor often splits the input stream into multiple blocks and resets the *codebook* (i.e., code word assignment table) at the beginning of each block. Figure 1 illustrates such an example. Even if the block length in the input stream may be fixed, compressed blocks generally have variable lengths.

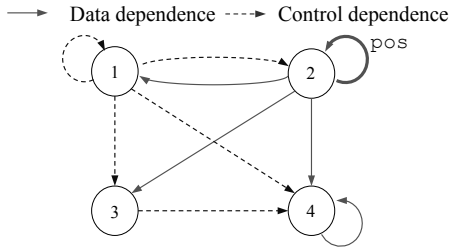
Table 1 surveys widely-used variable-length compression algorithms in various application domains. As the table illustrates, most of these algorithms are successfully parallelized at multiple granularities on the *compressor* side. However, parallel *decompressors* are much harder to find partly because it is difficult to extract efficient coarse-grain (e.g., block-level) parallelism. Although demanding less computations than the compressor, the decompressor is still an important target for parallelization as it often constitutes the critical path of a user application. This is particularly relevant on embedded/mobile platforms which cannot afford high-frequency cores due to cost and power constraints.

Figure 2 shows a simplified version of a variable-length decompression algorithm, which captures a common pattern across all the algorithms shown in Table 1. Every iteration of the loop decompresses one compressed block (Line 2). Position variable `pos` is updated to point to the starting position of the next compressed block, and decompressed data are stored into `out_buf`. Variables `eof` and `err` return end-of-file and error conditions, respectively. If no error occurred, the decompressed data are written to the output file (`out_stream`). This loop has three loop-carried dependences, as shown in Figure 2(b), which prevent parallel execution of multiple iterations. Line 1 creates a loop-carried control dependence through `eof`, Line 2 a loop-carried data dependence through `pos`, and Line 4 another loop-carried data dependence through `out_stream`.

Assuming high block count and very few erroneous blocks, the key challenge in parallelizing this decompressor loop is eliminating the loop-carried data dependence caused by the `pos` variable. Existing approaches address this by specializing the compressor to produce fixed-length compressed blocks by padding bits [8] and/or embedding hints from which the value of `pos` can be readily derived [18]. However, this approach has an obvious limitation of not working for a non-compliant input stream. Therefore, breaking this dependence without compressor-side support is crucial for practical block-level parallel decompression.

```
1: while (!eof)
   {
    // decompress one block
2:  len = decompress_block
          (&pos, out_buf, &eof, &err);
3:  if (!err && len>0)
4:      fwrite(out_buf, sizeof(char),
                len, out_stream);
   }
```

(a)



(b)

Figure 2: Example of variable-length decompression algorithm: (a) simple code; (b) program dependence graph.

2.2 Value Prediction-Based Speculative Parallelization

In general, it is difficult to find the block boundary (pointed to by `pos` in Figure 2) precisely without decompressing the preceding blocks. This makes speculative parallelization based on value prediction an attractive alternative. Instead of precisely calculating the block boundary, we may *predict* it at a much lower cost and decompress multiple blocks speculatively. Misspeculation detection can be performed by simply comparing the predicted starting position of a block with the ending position after decompressing the previous block. If they match, speculation is successful, and the program can continue execution; otherwise, it must rollback and re-execute the misspeculated block.

Some of exisiting speculative parallelization systems exploit value prediction to extract additional parallelism [21, 23, 26, 28, 32]. Most of the systems keep track of the value history of a variable of interest in the past to predict its value in the future. Common predictors include memoization predictor [21, 26], value stride predictor [28], and trace predictor [23]. However, the `pos` variable poses (nearly) zero correlations over time or across distinct inputs to make these predictors ineffective. Alternatively, a predictor function could be derived by *distilling* the original loop body down to an approximated version, which only computes the values of loop live-in variables [29, 32]. But, in most cases the complexity of calculating `pos` at the end of a block is comparable to that of decompressing the entire block since it requires the decoding of all code words within it. Moreover, complex dependence patterns and limited program analysis capabilities make it difficult for the automatic distiller to extract efficient, high-accuracy predictors in this domain.

Therefore, it would be highly valuable to provide a clean, easy-to-use API for a domain expert to implement a high-quality value predictor specific to a given variable-length decompression algorithm. This will allow the system to achieve much higher efficiency of parallel execution than domain-unaware speculative parallelization systems while minimizing programmer effort. This domain-specific framework based on value prediction is particularly attractive on resource-constrained embedded platforms. To realize this, three components are required:

- (Custom value predictors) Prediction algorithms to identify the starting point of each compressed block with high confidence and low overhead
- (Misspeculation detection and recovery mechanisms) Runtime support for misspeculation detection and recovery when speculation is incorrect
- (Parallelization API) Easy-to-use API to transform existing variable-length decompression algorithms into speculatively parallel codes based on value prediction

The remainder of this paper describes all of the above three components. Section 3 discusses prediction algorithms based on partial decompression and pattern matching. Section 4 introduces the SDM runtime system with its speculative pipeline execution model and mechanisms for misspeculation detection and recovery. Section 5 describes the parallelizing API of SDM with a code transformation example.

3. Block Boundary Prediction Algorithms

The prediction algorithms we propose are classified into two categories: *partial decompression*-based and *pattern matching*-based. Some compression algorithms define a block delimiter string that indicates the beginning of a new compressed block. Table 1 summarizes the existence of a delimiter pattern and its length for each algorithm. If the delimiter pattern does not exist or is short (e.g., fewer than 40 bits), partial decompression-based algorithms may be used; otherwise, simpler pattern matching-based ones are used. The former is applied to zlib (Section 3.1), and the latter to bzip2 and H.264 (Section 3.2).

3.1 Prediction Algorithm based on Partial Decompression

We first describe how to predict the starting point of a block encoded with dynamic Huffman coding via partial decompression. Then we apply it to identify well-aligned chunks in a zlib-compressed stream, which can be decompressed in parallel.

3.1.1 Prediction Algorithm for Huffman Coded Blocks

Huffman coding is a lossless entropy coding algorithm. Taking the frequency of appearance of each input symbol into account, the most frequent symbols are coded with shortest code words, whereas the least frequent symbols with longest code words. This encoding information is compressed in a tree format, called Huffman tree (or codebook). Static Huffman coding uses a single fixed codebook for the entire input, while dynamic Huffman coding produces an optimal codebook for each block. Henceforth, we will assume dynamic Huffman coding since static Huffman coding is just a special case of it.

Figure 3 illustrates the block structure of a Huffman-coded compressed stream. A Huffman tree is placed at the beginning of each block, following an optional delimiter string specific to a compression standard. The decompressor first reconstructs the Huffman tree from the compression block. Using this information, the rest of the compression block is decompressed. The end of the compression block is marked by a special code called the *End Of Block (EOB)*. This code is assigned to the longest code word in the codebook because it appears only once within each block.

We propose a block boundary prediction algorithm for Huffman-coded streams exploiting the EOB code with the following steps. First, the prediction algorithm extracts the EOB code of the block by decompressing the Huffman tree. Then a potential starting point of the next block is obtained by pattern matching with the EOB code. Note that there is a non-zero probability of false positives since the search is done by simple pattern matching without actu-

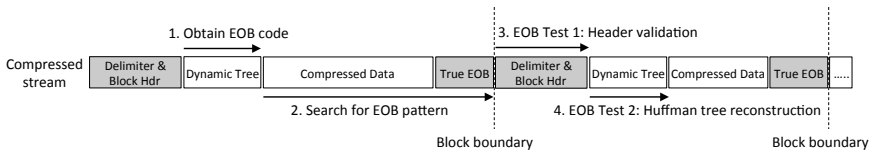


Figure 3: Identification of block boundaries from Huffman-coded compressed stream

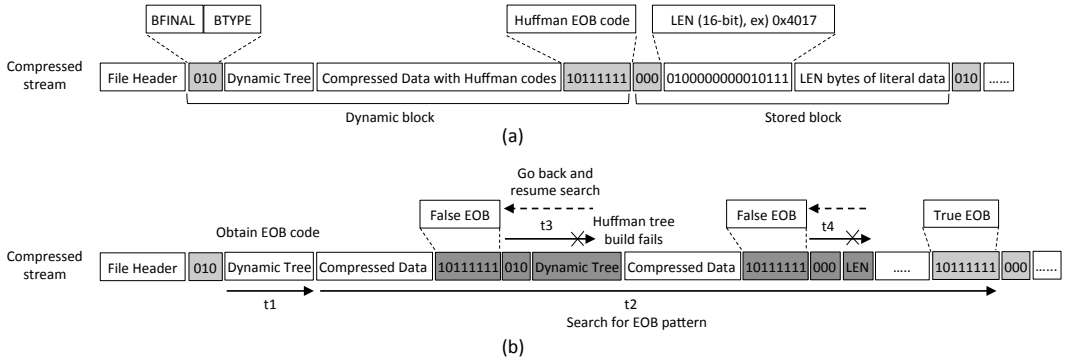


Figure 4: (a) Two interesting block types of zlib (`DYNAMIC`, `STORED`); (b) Operation of the prediction algorithm for `DYNAMIC` block

ally decompressing each code word. The probability increases for a shorter EOB code. To improve the accuracy of prediction, we employ two additional tests to validate an EOB:

- **Header validation**: By checking whether the header of the next block following the EOB code is well-formed or not, we can enhance the confidence of prediction. For example, if the compression algorithm of interest has a block delimiter string (as shown in Table 1), the predictor can make a comparison against the string. This can be applied to other header fields such as type, length, etc.
- **Huffman tree reconstruction**: Partial decompression of the next block also helps validating a true EOB. In most cases, a false-positive EOB causes an error while reconstructing a dynamic Huffman tree for the following block.

Although these additional tests significantly improve the accuracy, there is still a non-zero probability of false positives theoretically. Even so, program correctness is ensured by the misspeculation detection and recovery mechanisms discussed in Section 4.2.

3.1.2 Case Study: Zlib Decompression

Zlib (gzip) [15] implements the DEFLATE [15] compression algorithm, which is a combination of Huffman coding and the LZ77 algorithm [33]. Since it is based on Huffman coding, we can apply the prediction algorithm in Section 3.1.1 to identify block boundaries from a zlib-compressed stream. The rest of this section presents how we customize the prediction algorithm for zlib and augment it to break inter-block dependences caused by LZ77. Throughout this paper we use *minigzip* [15] as a reference implementation of gzip for its simplicity. However, the ideas introduced in this paper are generally transferable to other gzip implementations.

Figure 4(a) shows the file format of gzip. Each block starts with a 3-bit header containing two fields: `BFINAL` (1 bit) and `BTYPE` (2 bits). `BFINAL` indicates the end block. If `BFINAL` is set to "1", the block is considered as the last block of the compressed stream. `BTYPE` specifies the type of the block: (1) "00" indicates a *stored* block that directly copies the original data without compression. The following 16-bit `LEN` field contains the length of the uncompressed data; (2) "01" indicates a static Huffman-coded block; (3) "10" indicates a dynamic Huffman-coded block, which is typically the most common block type; (4) "11" is not used (invalid).

Figure 4(b) illustrates how the prediction algorithm in Section 3.1.1 works for a dynamic Huffman block (`BTYPE` = "10"). The prediction algorithm first decompresses the dynamic Huffman tree to obtain the EOB code word for the block. Then it searches for the EOB code by pattern matching. If the next match is a true EOB, it will pass the EOB validation tests and the search time will be a sum of time for EOB extraction (`t1`) and time for pattern matching (`t2`). The validation time with Huffman tree reconstruction is overlapped with the EOB extraction time (`t1`) for the next block. To reduce the EOB search time (`t2`), the scanner exploits minimum block length speculation; the scanner skips the first `min_blk_len` bytes after the Huffman tree and start an EOB search from there. The value of `min_blk_len` is set to 14,000 by default.

The validation test time for a false positive case increases the scanning time. Once an EOB pattern is found, the prediction algorithm first checks the validity of `BTYPE` of the next block. The valid patterns are "10" (`DYNAMIC`) and "00" (`STORED`). Note that static ("01") and dynamic ("10") Huffman blocks are not allowed to co-exist in a single compressed stream, so all practically interesting cases use dynamic Huffman coding. If the block type is `DYNAMIC`, a Huffman tree reconstruction test is performed. The time `t3` represents the overhead of this procedure. If the block type is `STORED`, the predictor tests the expected range of the `LEN` field. In minigzip [15], a stored block is emitted only when a 16 KB literal buffer is full and compression ratio with LZ77 is very small, hence the store block size is slightly greater than 16 KB in most cases. Therefore, the expected range is set to [16 KB, 16 KB+`speculated_range`] where `speculated_range` is 640 bytes by default. Note that, even if the value of `LEN` falls out of the range, correctness is preserved since it is handled as a misspeculation case.

The prediction algorithm described thus far enables us to efficiently obtain block boundaries predicted with high confidence. However, not all blocks in a zlib-compressed stream can be decompressed in parallel due to inter-block dependences created by the LZ77 algorithm, which employs a *dictionary-based* compression method. LZ77 compresses data by replacing repeated occurrences

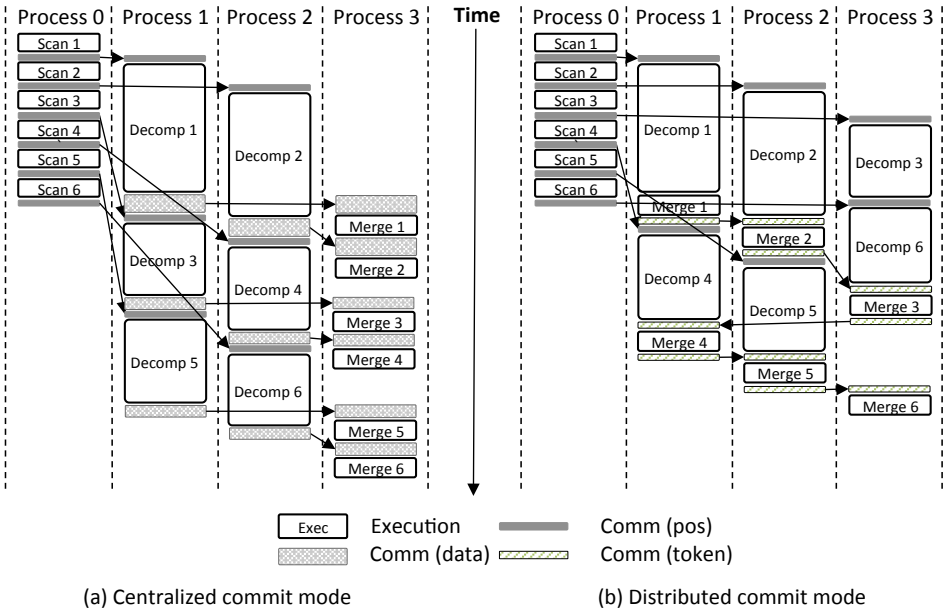


Figure 5: Two execution models of SDM : (a) Centralized commit; (b) Distributed commit

of a data pattern with a matching entry in the dictionary (referred to as a *sliding window*). The sliding window retains some of the recent, uncompressed input data already passed through the compressor. A match is encoded by a distance-length pair corresponding to the offset and length in the sliding window. Inter-block dependences may manifest through references to the sliding window to prevent parallel decompression of adjacent blocks. In minigzip, LZ77 uses a 32KB sliding window.

To put the decompressor into the right context (i.e., properly fill in the sliding window), we exploit stored blocks and overlapped chunk execution. Since a stored block is greater than 16 KB, two consecutive stored blocks will be large enough to fill in the 32 KB sliding window. This marks the beginning of an independently decompressible unit (IDU) spanning multiple blocks. One stored block fills only a half of the sliding window, so we need to redundantly decompress `N` extra blocks right before the stored block to fill in the remaining half. Since the `N` blocks are decompressed (using the INFLATE [15] algorithm) with an initially-empty sliding window, decompressed data do not yield correct results. This redundant decompression is performed only to fill in the sliding window; the correctly decompressed output will be gathered from execution of the previous chunk [2] which runs on a complete sliding window. According to our analysis using a number of gzipped files, the INFLATE algorithm works correctly if `N` is greater than 5.

We analyze the sensitivity of the prediction algorithm using 20 randomly picked packages from the list of Major Linux Application Programs compiled by the Linux Information Project [5]. The source code of each package is compressed using the *unmodified* minigzip with the default configuration parameters (Level 6). Those 20 packages are: abiword, dosemu, emacs, gftp, ghostscript, gqview, httpd, kaffeine, khexedit, lyx, mysql, perl, php, postgresql, python, samba, sql-ledger, tcl, xfig, xpdf, and zope. Our analysis indicates an average EOB length of 13.5 bits and 1.7 false-positive EOB matches per block. We find the two additional tests (header validation and Huffman tree reconstruction) to be essential to reduce the misspeculation rate, which effectively filter out *all* of these false positives. However, we also identify the following limitations and room for improvement:

- 10 out of the 20 gzipped packages have no stored blocks. For these files, the current implementation of the scanner cannot break inter-block dependences caused by LZ77, and an alternative method needs to be devised. The distribution of stored blocks also affects load balancing of parallel execution, which is discussed in Section 6.1.
- Three parameters control tradeoffs between prediction accuracy and performance: `min_blk_len`, `speculated_range`, and `N` (number of blocks overlapped between adjacent chunks). Although the default values for these parameters work well for minigzip, optimal settings might vary on other implementations of gzip. However, note that the setting of these values affects only performance but not correctness.

3.2 Prediction Algorithm based on Pattern Matching

Some compression algorithms define block delimiter patterns to improve error resilience against network errors. If the length of the delimiter is long enough (e.g., 40 bits or longer), the prediction algorithm can predict the starting point of a compressed block with high confidence by simple pattern matching. This approach is applicable to bzip2 and H.264.

The bzip2 file format defines a 48-bit pattern called *magic header* (`0x314159265359`), which signals the beginning of a new compressed block. In this case high-confidence prediction of starting point of block can be implemented by simple pattern matching. Unlike gzip, inter-block dependences do not exist in bzip2 because it performs block-based compression. Therefore, block-level parallel execution can be implemented easily.

The H.264 video compression standard also defines a special video frame that breaks all inter-frame dependences crossing it, called *Instantaneous Decoding Refresh (IDR)* access unit, which can be easily identified by a 40-bit type field (`0x0000000165`) of Network Access Layer Unit (NALU). Since no inter-frame dependences exist across an IDR frame, a group of video frames beginning with an IDR frame forms a chunk that can be independently decoded.

[2] A *chunk* is the unit of work dispatched to a worker process and consists of one or more IDUs.

4. SDM Execution Model

This section introduces the execution model of the SDM parallelization framework. To make best use of hardware resources in embedded platforms, SDM supports two parallel execution models: *centralized commit* and *distributed commit*. The former is more suitable for a platform with many available cores and high communication bandwidth, whereas the latter with a small number of cores and low communication bandwidth. We also describe low-cost misspeculation detection and recovery mechanisms in SDM.

4.1 SDM Three-Stage Pipeline

SDM implements speculative pipeline execution with three stages: Scanner, Decompressor, and Merger. Figure 5(a) shows the execution model with centralized commit. The SDM runtime system uses processes rather than threads for parallel execution contexts. At program invocation, the main process creates and configures each stage. The scanner process executes a prediction algorithm introduced in Section 3 and sends chunk boundaries to the next stage. The parallel decompressor processes decompress multiple chunks concurrently and send the results to the merger stage. The main process itself becomes the merger stage to commit speculatively decompressed data to the output file if no error occurred. Since the decompressor stage is responsible for most of useful work, we create as many decompressor processes as the number of available cores by default.

For manycore embedded platforms with a plenty of available cores, the *centralized* commit model takes full advantage of three-stage pipelining by allocating a separate process for the merger stage. In this model the decompressor and merger stages are fully overlapped. This model entails efficient communication between processes since decompressor-merger communication may become a performance bottleneck.

However, platforms with only a handful of cores may not benefit from this additional process due to context switching and memory space overhead. Hence, SDM also supports the *distributed* commit model to better support such platforms as shown in Figure 5(b). This model fuses the decompressor and merger stages to enable in-process commit across the decompressor processes. To preserve write order, a commit token is communicated among decompressor processes in a sequential execution order. If a decompressor process has finished processing a chunk but received the token yet, it stores the decompressed chunk in a process-local buffer and continue to decompress the next assigned chunk. The buffered data will be committed when the token is passed to the process. The distributed commit model may outperform the centralized commit model on more resource-constrained platforms.

4.2 Misspeculation Detection and Recovery

In case of misspeculation of block boundary, a mechanism that kills misspeculated execution and re-execute the program from previously known correct state must be provided to ensure correctness. Taking the compressed stream in Figure 1 for example, if Block 0 and Block 1 are executed in parallel, the ending state (i.e., value of `pos`) after processing Block 0 must match the speculated initial state before processing Block 1 (i.e., predicted value of $\texttt{pos}_1$).

The merger process is responsible for misspeculation detection and recovery. It compares the predicted `pos` value received from the scanner process with the corresponding, non-speculative `pos` value received from a decompressor process. Since the first `pos` value ($\texttt{pos}_0$) is guaranteed to be correct, all the following `pos` values can be verified inductively.

If a misspeculation is detected, all speculative scanner and decompressor processes are squashed, and the rest of the program is sequentially executed. It is our design decision not to restart parallel execution since the prediction algorithms we propose in Section 3 yield very high accuracy so that there is no performance benefit from recovery of parallel execution. However, it could be a useful addition when the misspeculation rate is high. We leave this for future work.

5. SDM API

The SDM API functions are classified into two categories as summarized in Table 3. The first category includes functions already implemented by SDM to initiate and orchestrate pipeline execution. The second category defines interface functions that encapsulate algorithm-specific prediction and decompression codes and must be implemented by the programmer.

Parameter	Type	Default	Description
num_process	int	# of cores	Number of decompressor processes
chunk_size	int	1	Chunk size in independently decompressible units
commit_mode	enum	CENT	Commit mode: CENT (centralized), DIST (distributed)
in_file	char*	Input file	Name of input file
out_file	char*	Output file	Name of output file

Table 2: `conf` structure provided by SDM

Configuration parameters for SDM execution are defined in `conf` structure shown in Table 2. `chunk_size` specifies the amount of work dispatched to a decompressor process at a time in independently decompressible units (IDUs). The IDU of bzip2 is a compressed block; the IDU of gzip is a group of blocks beginning with a stored block; the IDU of H.264 is an IDR-frame group.

5.1 Using the API: An Example

Figure 6 illustrates how the sequential variable-length decompression algorithm in Figure 2 is transformed into a parallel code using the SDM API. We assume centralized commit, and the lines taken from the original loop are shaded. The main function (Figure 6(d)) first creates scanner and decompressor processes and then enters into the `merge` function (Line 8).

Figure 6(a) shows the `scan` function that constitutes the first stage. `predict_boundary` is a programmer-defined function implementing a block boundary prediction algorithm. This function returns the starting position of the next chunk, which is sent to both decompressor stage (for decompression) and merger stage (for misspeculation detection). Figure 6(b) shows the `decompress` function, which runs in the second stage. This function receives a predicted position value (`pos`) from the scanner and invokes `decompress_chunk` to start decompression from the position for the next `chunk_size` IDUs. Then it sends to the merger process the decompressed data and the `pos` value after processing this chunk (`real_pos`) for misspeculation detection in the merger stage. Figure 6(c) shows the `merge` function for the merger stage, which performs misspeculation detection and recovery if necessary and writes to the output file.

5.2 Programmer-provided Functions

The programmer must provide the following two functions that encapsulate prediction and decompression algorithms requiring domain-specific knowledge. Function `predict_boundary` implements a prediction algorithm as discussed in Section 3. It takes input file name, starting position for search, and chunk size as inputs and returns the starting position of the next chunk. Function `decompress_chunk` performs actual decompression of `chunk_size` IDUs starting from the position indicated by `pos`. It returns the length of the decompressed data in bytes and sets `real_pos` to be used for misspeculation detection.

Operation	Description
SDM System-provided Functions	
`init(conf)`	Read file size and create queue. If the commit mode is DIST, allocate some buffer.
`destroy(conf)`	Destroy the created queue and buffers.
`create_process(function, &pid, arg)`	Create a new process with process id that will execute the function with the arg
`join_process(conf)`	Wait for worker processes with the configured number of processes
`produce(dst, &val)`	Enqueue val in software queue that send it to dst;
`consume(src, &val)`	Dequeue val from src
`scan(conf)`	Execute scan process; Forward a predicted starting point of block to the targeted process
`decompress(decomp_pid)`	Execute decompress process; compute the compressed data with predicted starting point of block
`merge(conf)`	Execute merger process; write decompressed blocks on output file (Centralized)
`recover_misspec(real_pos)`	Handle recovery from misspeculation
`insert_merge_list(output_buf, len, merge_list)`	Insert decompressed chunk to linked list; the decompressed chunks are stored to merge list for each process (Distributed)
Programmer-provided Functions	
`next_pos = predict_boundary(conf.in_file, pos, conf.chunk_size, &eof)`	Find the starting point of block in compressed stream by using prediction algorithm
`decompress_chunk(conf.in_file, pos, &err, &real_pos)`	Decompress a given chunk and compute the end point of block (real_pos)

Table 3: SDM runtime system interface

```
1  void scan (conf){
2      decomp_pid = 0;
3      while(!eof){
4          next_pos = predict_boundary(conf.in_file, next_pos, conf.
               chunk_size, &eof);
5          produce(decomp_pid, &next_pos);
6          produce(merge, &next_pos);
7          decomp_pid++;
8          if(decomp_pid == conf.num_process)
9              decomp_pid = 0;
10     }
11     for(decomp_pid=0;decomp_pid<conf.num_process;)
12         produce(decomp_pid++, EOS);
13 }
```

(a) Scanner

```
1  void decompress (decomp_pid){
2      while(TRUE){
3          consume(scan, &pos);
4          if(pos == EOS) break;
5          len = decompress_chunk (pos, out_buf, &eof, &err, &real_pos);
6          if(!err && len > 0){
7              produce(merge, &len);
8              produce(merge, out_buf);
9              produce(merge, &real_pos);
10         }
11     }
12     produce(merge, EOD);
13 }
```

(b) Decompressor

```
1  void merge (scan_pid, total_decomp, conf){
2      decomp_pid = 0;
3      while(TRUE){
4          consume(decomp_pid, &len);
5          if(len == EOD) break;
6          consume(decomp_pid, &real_pos);
7          consume(merge, &pred_pos);
8          if(real_pos != pred_pos)
9              recover_misspec(real_pos, scan_pid, total_decomp, conf);
10         consume(decomp_pid, out_buf);
11         fwrite(out_buf, sizeof(char), len, conf.out_stream);
12         decomp_pid++;
13         if(decomp_pid == conf.num_process)
14             decomp_pid = 0;
15     }
16 }
```

(c) Merger

```
1  void main(conf){
2      pid_t scan_pid, *total_decomp;
3      total_decomp = malloc(sizeof(pid_t) * conf.num_process);
4      init(conf);
5      create_process(scan, &scan_pid, conf);
6      for(i=0; i<conf.num_process; i++)
7          create_process(decompress, &total_decomp[i], i);
8      merge(scan_pid, total_decomp, conf);
9      join_process(scan_pid, total_decomp, conf);
10     destroy(conf);
11 }
```

(d) Main Process

Figure 6: Parallelizing sequential code in Figure 2(a) using SDM (centralized commit). (a) Scanner function (Stage 1); (b) Decompressor function (Stage 2); (c) Merger function (Stage 3); (d) Main function.

6. Evaluation

SDM is evaluated on two resource-constrained embedded platforms: Samsung Exynos 4412 quad-core platform (representing "fat" cores) and Tilera TILE-Gx8036 36-core platform (representing "thin" manycores). Table 4 shows the detailed specifications of the two platforms. Using SDM the latest versions of three popular production-grade decompression programs are parallelized: minigzip (zlib 1.2.7) [2], bzip2 (libbzip2 1.0.6) [1], and H.264 (JM-software 18.0) [3]. Three inputs taken from public domains are used for each program, as shown in Table 5. For H.264, an IDR-frame is inserted every 30 frames. The chunk size is set to one except for zlib (set to 5). Following the speedup measurement methodology of the SPEC CPU benchmark suite, we preload the input file into memory and write the decompressed output to a pre-allocated memory buffer without actually writing to the output file for both sequential and parallel codes.

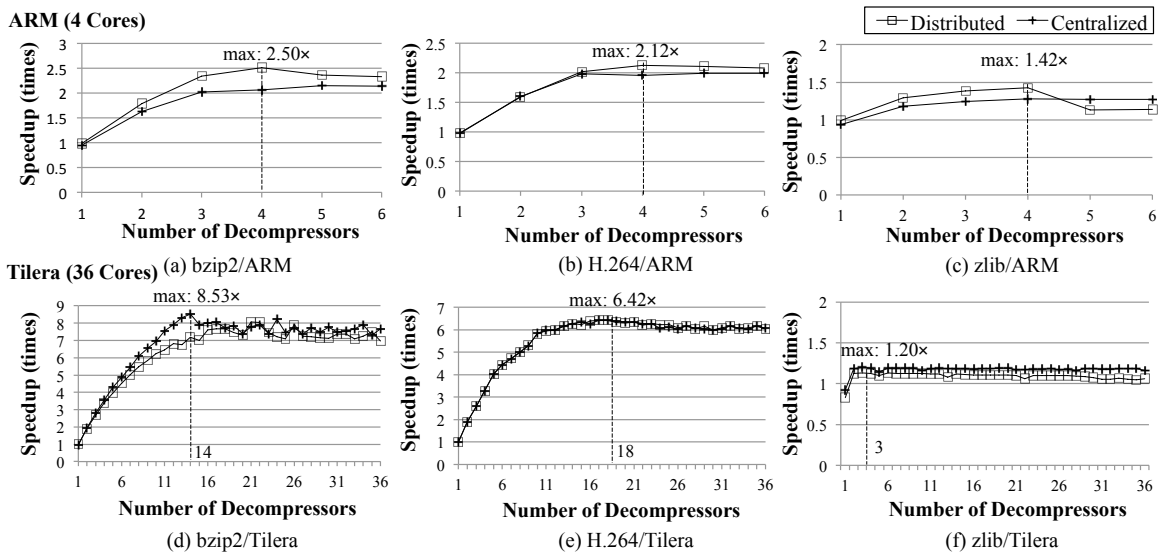


Figure 7: Whole program speedups calculated by taking geometric mean over three inputs for each data point on two platforms

	ODROID-X	NGSM-3600
Processor	Samsung Exynos 4412	Tilera Tile-Gx8036
# of cores	4 "fat" cores	36 "thin" cores
Core Speed	1.4 GHz	1.0 GHz
RAM	1GB, LP-DDR2	16GB, DDR3
OS	Linaro 12.10	Tilera Enterprise Linux 6.0
Compiler	gcc-4.4.6	gcc-4.6.3

Table 4: Detailed specifications for two embedded platforms

Decompression Utility	Input 1	Input 2	Input 3
bzip2 1.0.6 [1]	Firefox 1.7.13 [6]	Linux kernel 3.6.1 [11]	SPEC 2000 *ref* input [27]
zlib 1.2.7 [15]	Firefox 1.7.13 [6]	Linux kernel 3.6.1 [11]	SPEC 2000 *ref* input [27]
H.264-JM 18.0 [3]	Bridge 2000 frames [14]	Grandma 870 frames [14]	Akiyo 300 frames [14]

Table 5: Input files

6.1 Whole Program Speedups

Figure 7 shows the speedup results of the three algorithms on both platforms. The X-axis shows the number of decompressor processes, and the Y-axis speedups over the performance of the original sequential program compiled with `gcc -O3`. Each data point (for both sequential and parallel executions) is measured 10 times (excluding the time for input pre-loading and output buffer pre-allocation), and the minimum execution time is taken to isolate interferences from background processes. Then a geometric mean is calculated over the three inputs.

Figures 7(a)-(c) show whole program speedups for the three decompression programs on the fat quad-core platform. The maximum speedups are achieved with 4 decompressor processes: 2.50× for bzip2, 2.12× for H.264, and 1.42× for zlib. Beyond this point performance is saturated (or even goes down). Distributed commit performs consistently better than centralized commit for all three programs. We will discuss tradeoffs between the two commit modes in Section 6.2.

Figures 7(d)-(f) show the corresponding graphs on the thin manycore platform. The maximum speedups for the three programs

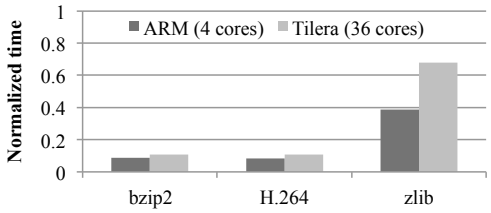


Figure 8: Scanner execution time normalized to execution time of the original sequential program

are achieved with different pipeline configurations: 8.53× with 14 decompressors for bzip2, 6.42× with 18 decompressors for H.264, and 1.20× with 3 decompressors for zlib. H.264 does not perform as well as bzip2 partly because two of the three input video clips have a relatively small size–Grandma has 870 frames (equal to 29 chunks), and Akiyo has only 300 frames (equal to only 10 chunks). Note that, unlike the quad-core platform, centralized commit performs comparably to or better than distributed commit.

On both platforms the performance of zlib is lower than the other two programs primarily due to two factors: higher scanner overhead and load imbalance. Figure 8 shows the first factor by comparing the scanner execution times of the three decompressors. As expected, the scanner stage of zlib employing partial decompression is heavier than pattern matching-based ones for bzip2 and H.264. According to Amdahl's Law, the maximum achievable speedup on the Tilera platform is upper bounded by 1.61×.

The second factor is load imbalance among parallel decompressor processes. As discusssed in Section 3.1, a new chunk can begin only on a stored block (i.e., `BTYPE` is `STORED` (00)). Since stored blocks are not uniformly distributed in general, load imbalance can occur. Table 6 evidences this. Table 6 tabulates speedups and load distributions across four decompressor processes for the three inputs. The third to sixth columns show how many bytes each process produces as a result of decompression. The most balanced input, `SPEC.gz`, has the smallest max/min ratio to achieve the best speedup of 1.80× on the quad-core platform. The most unbalanced input, `linux.gz`, has two big chunks decompressed by Processes 1 and 2; the other two processes are mostly idle to yield a much lower speedup of 1.24×. The third input, `firefox.gz`, falls somewhere between the other two inputs.

Input	Speedup	Process 1	Process 2	Process 3	Process 4
Firefox	1.29	87.26MB (40.7%)	41.80MB (19.5%)	19.41MB (9.1%)	65.66MB (30.7%)
Linux	1.24	302.49MB (64.8%)	163.35MB (35.1%)	0.32MB (0.1%)	0.15MB (0.0%)
SPEC 2000	1.80	14.16MB (22.0%)	14.77MB (23.0%)	24.88MB (38.9%)	10.19MB (15.9%)

Table 6: Speedups and load distributions for three gzipped inputs

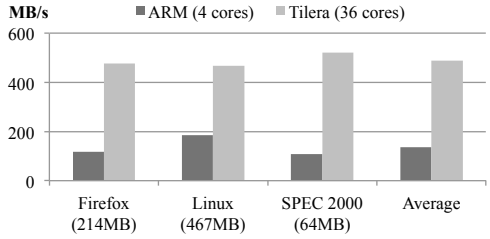


Figure 9: Communication bandwidth for two embedded platforms with centralized commit

6.2 Centralized Commit versus Distributed Commit

On the fat quad-core platform distributed commit mode generally performs better than centralized commit mode as shown in Figures 7(a)–(c). In contrast, on the thin manycore platform centralized commit performs comparably (for H.264 and zlib) or even better (for bzip2) as shown in Figures 7(d)–(f). This is because the fat-core platform has fewer high-throughput cores and lower inter-core communication bandwidth than the manycore platform.

To better understand the tradeoffs between the two commit modes, we measure inter-core communication bandwidth with MCRingBuffer [20] used for decompressor-merger communication and the three input files of bzip2. The results are shown in Figure 9, where the manycore platform yields consistently higher throughput than the quad-core platform. Centralized commit favors a platform with higher communication throughput (possibly with weaker cores), and distributed commit a platform with relatively lower communication throughput.

6.3 Misspeculation Recovery Overhead

We evaluate the efficiency of the SDM runtime system in handling misspeculations in Figure 10. Note that the prediction algorithms we introduce in Section 3 have very high accuracy, causing zero mispredictions. Therefore, we inject a mispredicted value at 25%, 50%, 75% and 100% locations from the beginning of the input file, where 100% indicates the end of the input file (i.e., no misspeculation). As mentioned in the Section 4, SDM will fall back to sequential execution from the point of misspeculation. We use bzip2 with the reference input from the SPEC CPU 2000 benchmark suite.

The results indicate that, although the speedup varies depending on where a misspeculation occurs, the SDM-parallelized bzip2 provides robust performance in face of misspeculations. If a misspeculation occurs near the beginning of the input file, the program suffers slight performance degradation due to the overhead of misspeculation detection and recovery as well as program initialization. However, in most cases, SDM performs faster than sequential execution. This demonstrates the efficiency of the SDM runtime in handling misspeculation cases.

6.4 Programmer Effort

Table 7 shows estimated programmer effort required to port the original sequential program to the SDM API with the number of

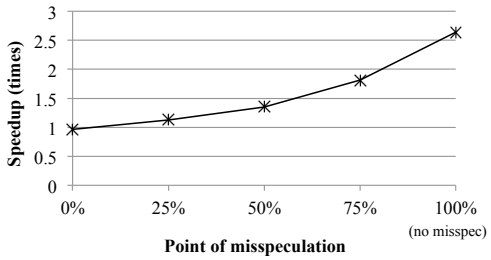


Figure 10: Misspeculation detection and recovery overhead

modified lines of code. A porting example is provided in Section 5.1. Columns 2 and 3 indicate the number of lines added for algorithm-specific prediction and decompression codes. As expected, the pattern matching-based boundary predictors for bzip2 and H.264 take fewer lines of code than the partial decompression-based predictor for zlib. Columns 4 through 6 measure the effort required to add to, modify and remove the existing code. Overall, it takes only modest effort to port to SDM since most changes happen within the two programmer-provided functions (`predict_boundary` and `decompress_chunk`) in an isolated manner.

7. Related Work

Parallel Huffman Decoding: Klein et al. [19] use the *self-synchronizing* property of Huffman coding to identify the boundaries of compressed blocks. Likewise, Zhao et al. [31] propose finite state machine (FSM) based speculative execution that performs the parallel decompression for multiple code words within a block. However, these techniques can be applied only to static Huffman coding, or within one block, where the codebook does not change. In theory, self-synchronization may never happen for particular inputs. Biskup et al. [17] introduce modifications to the original Huffman coding algorithm, which guarantees the occurrence of self-synchronization within a finite number of decompressed bits. This code word-level parallel decompression is complementary to SDM's block-level parallel decompression to exploit multi-level parallelism for a large block.

Parallelizing Variable-Length Video Decoders: Gurhanli et al. [18] parallelize the H.264 decoder in group-of-pictures (GOP) granularities, which has a similarity to this work. However, they embed hints for the starting point of GOP from the compression side. Thus, this approach has an obvious limitation of not working for a non-compliant input file. Nikara et al. [24] propose parallel variable-length decoding to decode multiple code words in parallel in MPEG-2 video streams. However, their technique is applicable to a single frame (block) and uses custom hardware based on FPGA. Bilas et al. [16] propose pipeline execution to implement real-time MPEG-2 video decoder, which exploits GOP-level parallelism via pattern matching. SDM is a more general framework that targets the domain of variable-length decompressors and uses a predictor based on partial decompression as well.

Speculative Parallelization Frameworks: Raman et al. [26] propose speculative parallel iteration chunk execution (Spice) parallelization model, which exploits a memoization predictor on hardware. Master/slave speculative parallelization (MSSP) [32] and the adaptive multiple value prediction scheme by Tian et al. [29] automatically extract one or more software value predictors for loop live-ins by taking backward data slices of the original loop. However, all of these hardware and software predictors have limited accuracies. On embedded platforms with limited resources the accuracy and efficiency of the predictor are crucially important to jus-

Program	Programmer-provided functions		Modification of existing code			Total lines modified	Total lines of original program
	predict_boundary()	decompress_chunk()	Added	Modified	Deleted		
bzip2-1.0.6	15	15	37	2	6	75	3147
H.264-JM 18.0 (decoder)	7	22	17	0	4	50	20718
zlib-1.2.7	142	76	26	0	10	254	4055

Table 7: Programs enhanced using SDM. Columns 2-7 indicate the programmer's effort required to port the original sequential program to the SDM API. Column 8 is the lines of the original source code.

tify the cost of custom predictors introduced in this paper. There are runtime support systems for speculative pipeline parallelization such as CorD [30] and SMTX [25], which might be used to implement a system similar to SDM. However, SDM elevates the abstraction level of the API, specifically targeting the domain of variable-length decompression to better separate the expertise of the algorithm from that of parallelization. Also, SDM is specialized to run more efficiently with a handful of cores via distributed commit, removal of memory version tracking, and so on. Mankin et al. propose a transactional memory system that runs efficiently on embedded systems [22]. But, this system is suitable only for speculative DOALL parallelization where iterations are mostly independent without frequently arising cross-iteration dependences.

8. Conclusion

This paper advocates value prediction-based speculative parallelization to effectively parallelize variable-length decompression algorithms, which would otherwise be difficult to parallelize. For this, we propose low-cost prediction algorithms based on partial decompression and pattern matching, to quickly identify block chunks that can be independently decompressed. Also, the SDM API and runtime system is designed and implemented to streamline parallel decompression of multiple chunks. The SDM API provides a clean, easy-to-use interface for an algorithm expert to implement a high-quality value predictor specific to a given variable-length decompression algorithm. The SDM runtime is specialized to execute this pipeline efficiently on resource-constrained embedded platforms while ensuring correctness by detecting and recovering from misspeculations. We have demonstrated the practicality of SDM by successfully parallelizing zlib, bzip2, and H.264 with only modest programmer effort. When evaluated on fat quad-core and thin 36-core embedded platforms, the SDM parallelized code achieves maximum speedups of 2.50× and 8.53× (and geometric mean speedups of 1.96× and 4.04×), respectively.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and Albert Cohen for shepherding this paper. We also thank Hanjun Kim and Nick Johnson for reviewing an early draft of this paper, Jae Young Jang for providing an optimized communication queue, Kwang Hyun Won for his help with the H.264 decoder, and Narinet Inc. for their support with the Tilera machine. This work was supported in part by the IT R&D program of MKE/KEIT [KI001810041244, Smart TV 2.0 Software Platform] and a research grant from Samsung Electronics.

References

[1] bzip2 and libbzip2. `http://bzip2.org/`.

[2] gzip homepage. `http://www.gzip.org/`.

[3] H.264: Advanced video coding for generic audiovisual services. `http://www.itu.int/rec/T-REC-H.264/`.

[4] JPEG homepage. `http://www.jpeg.org/jpeg/`.

[5] The Linux Information Project. `http://linfo.org/`.

[6] Mozilla Developer Network. `https://developer.mozilla.org/`.

[7] Parallel bzip2. `http://compression.ca/pbzip2/`.

[8] A parallel implementation of gzip. `http://zlib.net/pigz/`.

[9] Portable Network Graphics. `http://www.libpng.org/pub/png/`.

[10] Samsung Exynos 4 Quad. `http://www.samsung.com/exynos/`.

[11] The Linux Kernel Archives. `http://www.kernel.org/`.

[12] Tilera TILE-Gx processor family. `http://www.tilera.com/`.

[13] Vorbis audio compression. `http://xiph.org/vorbis/`.

[14] YUV CIF reference videos. `http://trace.eas.asu.edu/yuv/`.

[15] zlib: A massively spiffy yet delicately unobtrusive compression library. `http://zlib.net/`.

[16] A. Bilas, J. Fritts, and J. P. Singh. Real-time parallel MPEG-2 decoding in software. In *Proc. of IPPS*, 1997.

[17] M. T. Biskup. Guaranteed synchronization of Huffman codes. In *Proc. of Data Compression Conference (DCC)*, 2008.

[18] A. Gurhanli, C. C.-P. Chen, and S.-H. Hung. Coarse grain parallelization of H.264 video decoder and memory bottleneck in multi-core architectures. *International Journal of Computer Theory and Engineering*, 2011.

[19] S. T. Klein and Y. Wiseman. Parallel Huffman decoding with applications to JPEG files. *Computer Journal*, 2003.

[20] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Proc. of IPDPS*, 2010.

[21] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proc. of PPoPP*, 2006.

[22] J. Mankin, D. Kaeli, and J. Ardini. Software transactional memory for multicore embedded systems. In *Proc. of LCTES*, 2009.

[23] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *Proc. of ISCA*, 1999.

[24] J. Nikara, S. Vassiliadis, J. Takala, M. Sima, and P. Liuha. Parallel multiple-symbol variable-length decoding. In *Proc. of ICCD*, 2002.

[25] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proc. of ASPLOS*, 2010.

[26] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *Proc. of CGO*, 2008.

[27] Standard Performance Evaluation Corporation. `http://www.spec.org/`.

[28] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, 2002.

[29] C. Tian, M. Feng, and R. Gupta. Speculative parallelization using state separation and multiple value prediction. In *Proc. of ISMM*, 2010.

[30] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proc. of MICRO*, 2008.

[31] Z. Zhao, B. Wu, and X. She. Speculative parallelization needs rigor: Probabilistic analysis for optimal speculation of finite state machine applications. In *Proc. of PACT*, 2012.

[32] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. of MICRO*, 2002.

[33] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, Sept. 2006.