Fast Huffman decoding by exploiting data level parallelism

Citation for published version (APA):
Drijvers, T., Alba Pinto, C. A., Corporaal, H., Mesman, B., & Braak, van den, G. J. W. (2010). Fast Huffman decoding by exploiting data level parallelism. In F. J. Kurdahi, & J. Takala (Eds.), *Proceedings of the 2010 International Conference on Embedded Computer Systems (SAMOS), 19-22 July , 2010, Samos Greece* (pp. 86-92). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/ICSAMOS.2010.5642080

DOI:
10.1109/ICSAMOS.2010.5642080

Document status and date:
Published: 01/01/2010

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Fast Huffman Decoding by Exploiting Data Level Parallelism

Tim Drijvers*†, Carlos Alba Pinto†, Henk Corporaal*, Bart Mesman*, Gert-Jan van den Braak*
*Department of Electrical Engineering
Eindhoven University of Technology, The Netherlands
Email: {h.corporaal,b.mesman,g.j.w.v.d.braak}@tue.nl
†Silicon Hive
Eindhoven, The Netherlands

***Abstract*—The frame rates and resolutions of digital videos are on the rising edge. Thereby, pushing the compression ratios of video coding standards to their limits, resulting in more complex and computational power hungry algorithms. Programmable solutions are gaining interest to keep up the pace of the evolving video coding standards, by reducing the time-to-market of upcoming video products. However, to compete with hardwired solutions, parallelism needs to be exploited on as many levels as possible. In this paper the focus will be on data level parallelism. Huffman coding is proven to be very efficient and therefore commonly applied in many coding standards. However, due to the inherently sequential nature, parallelization of the Huffman decoding is considered hard. The proposed fully flexible and programmable acceleration exploits available data level parallelism in Huffman decoding. Our implementation achieves a decoding speed of 106 MBit/s while running on a 250 MHz processor. This is a speed-up of $24\times$ compared to our sequential reference implementation.**

I. Introduction

Modern video coding standards are becoming more computational power, bandwidth and hardware resource demanding. These standards are driven by an active research area in displays, a requiring consumer market and a still rising number of videos on the Internet. Providing faster frame rates and larger resolutions in a space efficient manner is key to success of these coding standards. New optical storage systems, such as Blu-ray and HD-DVD, are introduced to cope with the call for more bandwidth and storage space, thereby also pushing the compression ratios of video coding standards to their extend, resulting in more complex and computational power hungry algorithms.

These rapidly evolving video coding standards drive the industry more towards programmable solutions. Offering the ease of programmability to handle these quick changes in coding standards. To compete with hardwired solutions, programmable solutions exploit parallelism on many different levels to fulfill the request for computational power; examples are: task, instruction and data level parallelism, available in multi-core systems, very long instruction word (VLIW) processors and vector processors respectively. Media vector processors, such as the Silicon Hive VSP 2200 [1] and Stanford's Merrimac [2], have been introduced to cope with the increasing performance demands of real-time decoding of these modern video standards. Using a combination of VLIW and SIMD offers a great deal of parallelism on both an instruction level as well as on a data level.

One of these emerging video coding standards is Microsoft's VC-1, a mandatory video codec for Blu-ray, HD-DVD and growing in popularity on the Internet. The challenge taken in this work is to drastically improve the speed of the bitstream parsing of VC-1, which needs to be able to process at least 45 MBit/s. In order to be applicable to other video and imaging coding standards as well, we aim at a fully programmable solution.

Huffman coding [3], as applied in VC-1, assigns bit-lengths proportional to the frequency of use, resulting in high compression factors. These encoded bits, their length and the original symbol are typically stored in a Huffman table. Since the encoded symbols no longer reside within a fixed number of bits, the decoding process becomes more complicated. The next symbol can not be read from the input bitstream before the bit-length of the previously coded symbol has been determined. Huffman decoding therefore consists out of two operations, code-length determination and retrieval of the original decoded symbol as shown in Fig. 1.

In modern video coding standards, the structure of bitstreams are becoming more complex. To further improve the compression many different Huffman tables are used. The choice between these tables depends on the specific state of the decoder (e.g. decoding ac-coefficients, motion-vectors, etc.). For the VC-1 standard we distinguish over 100 different Huffman tables [4].

It has been stated by recent articles [5], [6] that the use of SIMD would be ineffective to optimize the speed of Huffman decoding, due to data dependencies. In this work we present a technique that allows to exploit data level parallelism that is available in Huffman decoding. This allows the use of SIMD

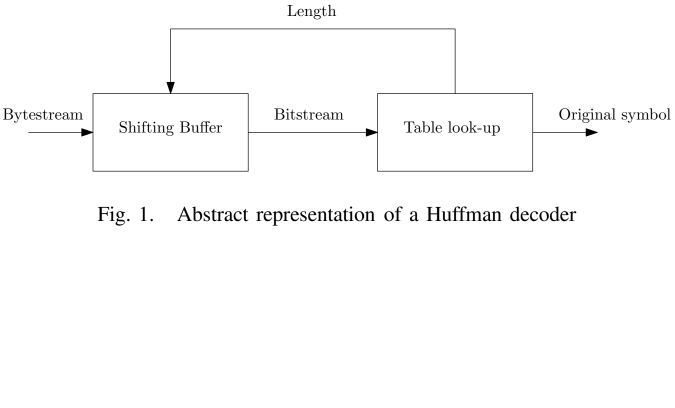

Fig. 1. Abstract representation of a Huffman decoder

978-1-4244-7938-2/10/$26.00 ©2010 IEEE

instructions in order to accelerate the Huffman decoding, while maintaining full flexibility of the Huffman tables without additional computational expensive modifications of those tables and full processor programmability.

This paper is organized as follows. First a range of related work is discussed, in which proposed programmable solutions are considered. Continuing with a more in-depth explanation on Huffman decoding. After this we discuss our data parallel Huffman decoding technique, followed by experimental results. Concluding this paper with an analysis on speed, energy and area.

II. Related work on programmable solutions

Within the field of Huffman decoding, various approaches have been taken to accelerate the decoding process in a programmable solution. What can be seen from Fig. 1 is the fact that the table look-up process and shifting buffer form the critical path.

A common optimization is the one proposed in [7], here a co-processor with one issue slot is capable of performing Huffman decoding and bitstream parsing. They use a 'sequencer' (hardwired converter from bytestream to bitstream) and hardwired Huffman tables to meet their performance. However, this solution offers very little flexibility.

A range of instruction-set extensions for a RISC have been presented in [6], supporting multiple video coding standards while maintaining programmability. However the Huffman tables need to be modified on forehand, calculating the number of leading zeros and perform clustering based on this information. In [5] a range of possible optimization techniques are proposed, taking similar approaches by grouping based on leading zeros.

More instruction-set extensions are proposed in [8] and [9]. Where the former proposes some software optimizations (loop transformations and clever alignment of lookup tables) and a new instruction (based on a barrel-shifter) for a TriMedia/CPU64. Hence the bitstream parsing and table lookup are somewhat optimized. The latter proposes a series of new instructions to speed-up the bitstream parsing (getbits, showbits, flushbits), however no optimizations are proposed for the table lookup process.

Yet an other extension made to the TriMedia/CPU64 is based on a FPGA-functional unit [10]. This functional unit is capable of decoding a single Huffman symbol in a 8 cycle function, resulting in a 43% improvement. However the proposed solution lacks real flexibility, since the FPGA can hold a limited number of table entries and run-time reconfiguration is difficult. The 43% improvement is insufficient in our case, especially considering the size of the FPGA that is required.

The multi-layer prefix grouping technique for parallel Huffman decoding in [11] describes a novel implementation of the Huffman algorithm on a VLIW processor, instead of an extension to the instruction set. It is implemented as a two-level lookup approach, which solves long decoding cycles and table size explosion at the same time. Their solution makes the number of clock cycles needed for decoding independent on the codeword length. Also in this approach the 89% improvement in performance is still insufficient for our goals, but full flexibility is retained.

A final work worth mentioning is the widely discussed single-side growing Huffman table (SGHT), proposed by Hashemian in [12]. A clustering algorithm is used to avoid the high sparsity of a Huffman table, resulting in a SGHT that can be stored efficiently. Moreover, due to the arithmetic properties of the SGHT and the encoded bitstream, a symbol can be decoded using simple arithmetic operations. However, this technique results in a SGHT with different code-words than the original coded Huffman table and an optimal solution for the clustering is still an open problem.

None of these proposed solutions allows fast Huffman decoding while maintaining full flexility of the Huffman tables and programmability of the parsing processor. This is highly desirable when aiming at a solution that supports multiple standards. For example, Huffman tables in JPEG are encoded in the header, this requires flexibility of the Huffman tables. In the VC-1 video coding standard more than 100 different Huffman tables are used, resulting in a complex bitstream where one requires fast switching between these tables.

TABLE I
Example Huffman table

Symbol:	Frequency:	Bitcode:	Length:
space	7	111	3
a	5	010	3
e	3	000	3
f	3	1101	4
t	2	1010	4
h	2	1000	4
i	2	0111	4
s	2	0010	4
l	2	1011	4
m	2	0110	4
n	2	11001	5
o	1	00110	5
p	1	10011	5
b	1	11000	5
u	1	00111	5
x	1	10010	5

III. Huffman decoding

Table I shows an example Huffman table generated for the input "this is an example of a Huffman table". The table shows the symbols used in the input, the frequency of use of these symbols, the assigned bitcodes and the length of these codes in bits. We assume that the table is sorted descending by frequency of use of the symbols.

Let us describe each column of the Huffman table as a list, where $column[i]$ holds the i-th element of $column$ indexed starting from 0. Here $column$ is either $symbol$, $bitcode$, or $length$. In our example $symbol[0]$ is 'space'. The frequency

```
Require: Next 32 bits from input stream $s$
Require: Length of the Huffman table $len$
  for $i = 0$ ; $i < len$ ; $i = i + 1$ do
    if $hit(s, bitcode[i], length[i])$ then
      return ($symbol[i]$,$length[i]$)
    end if
  end for
  return Error
```

Fig. 2. Linear search over the Huffman table

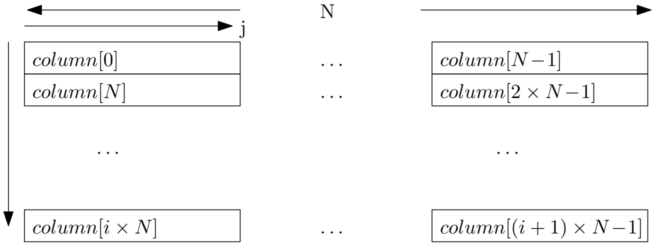

Fig. 3. Alignment of vectors and their elements in memory.

column is not required for decoding, but merely shown for an illustrative purpose. Secondly a bitstring is represented as a binary number (e.g. 10110101_2).

The $bits(w, l)$ function takes the upper l bits from the input word w, for example $bits(10110101\ldots00_2, 4) = 1011_2$, similar to the getbits operation proposed in [9]. The $hit(s, b, l)$ function returns $True$ when a hit is found with the given input. Here s contains the next 32 bits of the input bitstream, b and l contain the bitcode and the length of the current table row. For example: $hit(10110101\ldots00_2, 1011_2, 4) = True$, which is in fact: $hit(10110101\ldots00_2, bitcode[8], length[8]) = True$. So we have a hit and should return the symbol $symbol[8]$ which is 'l', according to our example Huffman table.

Since the Huffman table is sorted descending by frequency of usage of the coded symbols, a linear scan from the top of the table to the bottom is likely to find a hit within the first rows of the table. For example in table 130 of the VC-1 video coding standard, there is a probability of 74% that a linear scan will find a hit within the first 10 rows.

IV. Exploiting data level parallelism

The algorithm in Fig. 2 shows a linear table search. This algorithm is highly conditional, therefore it is not suitable to apply techniques such as software pipelining in order to increase the ILP (instruction level parallelism). ILP is a common level of parallelism available in VLIW processors.

The performance of a VLIW processors can be improved in two obvious ways: increasing the clock frequency or increasing the number of issue slots. However, a third improvement is available by using single instruction multiple data (SIMD) vector issue slots. In a N-way SIMD vector issue slot, a single instruction is simultaneous performed on N vector elements. These operations can be performed in two ways: either inter-vector or intra-vector. In inter-vector operations the same operation is element-wise performed on multiple vectors. Intra-vector operations are applied within the same vector on N elements.

The basis of this approach lies within speculative 'look-ahead' searching, reducing load operations, data packing and reducing the conditional behavior of the algorithm. First of all, let us discuss the base of our SIMD approach in which columns of the Huffman table are split into multiple vectors with an equal length N, rather than splitting columns into cells which are basically vectors of length 1. The final vector is padded with stuffing values until it meets the length N, we will refer to this padding as table gaps.

$$\overrightarrow{column_i} = (column[i \times N], \ldots, column[i \times N + N - 1])$$

Here i represents the i-th vector of column $column$ and N the chosen N-way, in our case $N \in \{8, 16, 32\}$. We can address an element in the vector as follows $\overrightarrow{column_i}[j]$ which is $column[i \times N + j]$, here j represents the j-th element from this vector ($0 \leq j < N$). Furthermore let us define a flag to be a vector of booleans with length N. In Fig. 3 the alignment of the cells of a Huffman column in memory are shown. A single access to this memory on address i results in a vector $\overrightarrow{column_i}$ containing a row of N elements.

Now, lets introduce the vector variant of our *hit* operation: $\overrightarrow{flag}_i = hit_v(s, \overrightarrow{bitcode}_i, \overrightarrow{length}_i)$. This operation is an intra-vector, performing element wise operations on the vector input. Defining the semantics of this function as follows:

```
for $j$ from 0 till $N - 1$ do
  $\overrightarrow{flag}_i[j] = hit(s, \overrightarrow{bitcode}_i[j], \overrightarrow{length}_i[j])$
end for
```

Taking a SIMD approach, each processing element j in the processor's vector datapath executes the *hit* function on the corresponding data element j. To be more precise the first processing element executes the *hit* function on $\overrightarrow{bitcode}_i[0]$ and the last element on $\overrightarrow{bitcode}_i[N-1]$ (hence the alignment of vectors in Fig. 3).

A correct Huffman table guarantees decompression uniqueness, this means that no code is a prefix of another. Due to this property we know that we will find at most one hit in the Huffman table. Therefore we can state that there will be at most one True value in the $\overrightarrow{flag}_i$ vector. In order to check if the 'flag is true', we sum all elements using a bitwise-OR using the function $found$ as defined below. If $found(\overrightarrow{flag}_i)$ is True there is at least one True value in the vector $\overrightarrow{flag}_i$, including decompression uniqueness implies there is at most one True value.

$$found(\overrightarrow{flag}_i) = \bigvee_{0 \leq j < N} \overrightarrow{flag}_i[j]$$

To find the corresponding value when the flag is true, a second operation is introduced: $pick_v(\overrightarrow{flag}_i, \overrightarrow{column}_i)$, with semantics defined as follows. This operation is an inter-vector operation, mapping a vector to a scalar. An example of

```
Require: Next 32 bits from input bitstream $s$
  $\overrightarrow{flag} = (0, \ldots, 0)$
  $i = 0$
  while not $found(\overrightarrow{flag})$ do
    $\overrightarrow{flag} = hit_v(s, \overrightarrow{bitcode}_i, \overrightarrow{length}_i)$
    $i = i + 1$
  end while
  return $(pick_v(\overrightarrow{flag}, \overrightarrow{symbol}_{i-1}), pick_v(\overrightarrow{flag}, \overrightarrow{length}_{i-1}))$
```

Fig. 4. Data parallel search over the Huffman table

applying this pick function: $pick_v((1, 0, 0 \ldots, 0), \overrightarrow{symbol}_0) =$ 'space'.

```
for $j$ from 0 till $N - 1$ do
  if $\overrightarrow{flag}_i[j]$ then
    return $\overrightarrow{column}_i[j]$
  end if
end for
return 0
```

Additionally, we can further reduce the conditions in the linear search algorithm (Fig. 2) by assuming bitstream correctness, stating that the input bitstream is correctly coded according to the Huffman table. Applying this property to the data parallel algorithm allows us to remove the loop over the length of the table, because we know that there exists a hit in the table. The algorithm now only contains one conditional statement (i.e. the while).

We modify our linear search algorithm to use these vector functions and assumptions on the Huffman table and bitstream, Fig. 4 shows the modified algorithm.

A. Further improvements

Performing the parallelization as proposed in Fig. 4 still yields at least three load instructions in a load-store architecture, assuming that the Huffman tables are not stored inside the issue slot (we want this to maintain full and easy programmability, even during runtime). In order to reduce this overhead we pack the three vectors into a single vector, hence it requires only one load now. Instead of loading three separate column vectors, one 'big' vector will be loaded. Now each vector element contains three packed fields, we can say that $\overrightarrow{packed}_i[j]$ is a triplet of $\overrightarrow{bitcode}_i[j]$, $\overrightarrow{length}_i[j]$ and $\overrightarrow{symbol}_i[j]$.

Increasing the speed even further, we introduce a Huffman load unit. This unit loads the vector, applies the *hit* function and post-increments the counter i, this results in a 2-cycle pipelined function. The advantage of this approach is that we do not need to wait for the load of the vector data to finish before we apply the *hit* function. These modifications result in the following algorithm, as shown in Fig. 5.

V. Experimental results

Experiments have been performed on artificial inputs to test this new idea, doing so we used table 130 of the VC-1 video coding standard, this is a 126 entry Huffman table. We generated inputs based on the average probability distribution as observed in a range of well known example test movies (the foreman, claire, suzie sequences), making this an input with realistic distribution of symbols. Doing so, we create an environment in which we can closely observe the result of our improvements. Using the table distribution as observed by a range of sample input movies, there is 74% chance that an encoded symbol is found within the first 10 rows of the Huffman table, this fact forms the base for the achieved acceleration.

```
Require: Next 32 bits from input bitstream $s$
  $\overrightarrow{flag} = (0, \ldots, 0)$
  $i = 0$
  while not $found(\overrightarrow{flag})$ do
    $(\overrightarrow{flag}, \overrightarrow{symbol}, \overrightarrow{length}, i) = huffman(s, i)$
  end while
  return $(pick_v(\overrightarrow{flag}, \overrightarrow{symbol}), pick_v(\overrightarrow{flag}, \overrightarrow{length}))$
```

Fig. 5. Data parallel search over the Huffman table, using a combined load and Huffman operation

A. Target platform

As programmable platform we have chosen for a Silicon Hive stream processor (SP). Silicon Hive cores are VLIW processors, which can contain a vector datapath allowing SIMD operations. Silicon Hive processors and systems are flexible during design time and programmable when ready. Silicon Hive offers a simulation environment in which the applications run on models of the cores and systems. The behavior of the processor's and system's composition can be changed without big effort, providing an easy environment to explore our design space.

Silicon Hive processors are described in a high level hardware design language called The Incredible Machine (TIM) [13]. In the TIM language a processor can easily be described, ranging from register files, interconnections to the semantics of functional units. TIM can easily be translated into VHDL or Verilog by assembling prewritten blocks of hardware. Moreover, TIM forms the input for the Silicon Hive compiler and simulation toolchain.

The Silicon Hive SP is a three issue slot scalar VLIW processor, with a word size of 32-bits. The processor contains special instructions for bitstream parsing, which allow reading and viewing of a number of bits from a scalar input, as proposed in [9]. This improves efficiency of the bitstream parsing, since the processor has a 32-bit datapath. Moreover, 2-way SIMD functional units are available that operate on normal scalar input data (i.e. the elements in the vector part have width 16 bit), this is especially useful for processing of e.g. motion-vectors (containing a X and Y component). We assume that the processor is running at a 250 MHz clock-rate.

Four custom operations are used, namely: *getbits*, a three cycle pipelined operation that reads a number of bits from the bitstream; *peekbits* is a two cycle pipelined operation that reads a number of bits from the bitstream, without changing the bitstream; *huffman* a two cycle pipelined SIMD operation that performs a load from memory and applies the hit_v

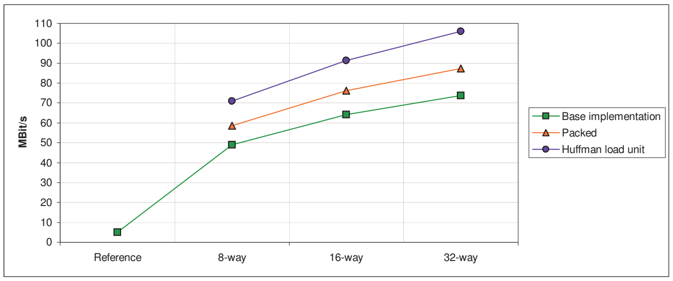

Fig. 6. Huffman decoding speeds in MBit/s

function; *pick* is a single cycle SIMD inter-vector operation that translates a vector into a single scalar based on an input flag.

B. Reference implementation

As a reference an implementation has been made based on the linear search algorithm shown in Fig. 2. Furthermore, the bitstream operations *getbits* and *peekbits* are used. Finally no modifications are made to the data path and memory system of the processor. Since the SP's datapath is 32 bits wide, each cell of the Huffman table consumes 32 bits.

C. Performance results

Let us consider the speed-ups gained by three implementations of our idea, we tested the vectorization with a N-way of 8, 16 and 32. The results of these experiments are listed in Fig. 6. One can see we reached a top speed of 106 MBit/s with a 32-way data parallel approach. As earlier mentioned three different approaches have been implemented. The base implementation shows the results of simply applying the proposed vectorization technique, secondly we applied packing of the three columns (i.e. bitcode, length, symbol) into a single packed column and finally the Huffman load unit is tested.

D. Table gaps

In this section we analyze the memory consumption of the data parallel approach that has been proposed. This analysis is performed on a real life example, namely the VC-1 video coding standard, this standard uses over 100 different Huffman tables. Let us first introduce the principle of table gaps.

Since vectors need to be aligned in our memory and Huffman table entries are not always a multiple of N, gaps are introduced in between the vectorized tables. Tables need to be aligned, because for every different mode of the encoder a different table is required. Therefore these tables need to be stored strictly separated in memory. A vector at the end of a table with a length smaller than N introduces gaps. The size of the introduced gap is defined as follows, here $\oplus$ is a module operation.

$$overhead(n) = \begin{cases} 0 & \text{if } (n \oplus N) = 0 \\ N - (n \oplus N) & \text{if } (n \oplus N) \neq 0 \end{cases}$$

Getting the total gap size for a given table can be calculated by taking the length of the table as input, $overhead(length)$.

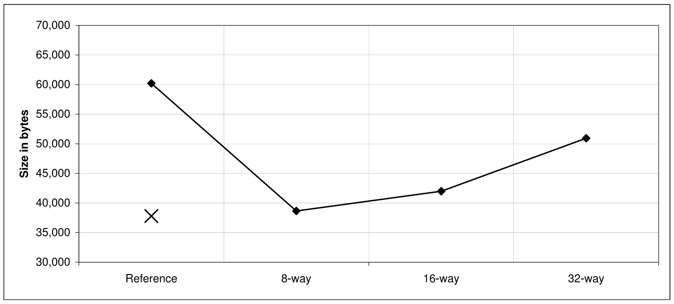

Fig. 7. Memory consumption in bytes of the reference approach and data parallel approach

The total size of the table, taking the gaps into account is shown in the Fig. 7, moreover this table also shows the reference implementation. The element size per vector is 64 bits, containing *bitcode*, *length*, *symbol* respectively consuming 24, 8 and 32 bits. Each cell in the reference code consumes 32 bits (because of the width of the scalar memory), hence a row consumes 96 bits. Therefore a reduction in memory, in terms of bytes, is seen in the figure. In order to show a more fair overview we also calculated the size in bytes for a possible packed version, this size is denoted by the cross in the graph.

VI. Energy

Due to the speculative behavior of the proposed acceleration technique, more energy is consumed compared to the sequential reference implementation. For example, when a hit is found at the first table element: $N - 1$ table elements can be considered overhead, as also the application of $N-1$ times the hit operation. We refer to energy loss as the percentage of unnecessary applied hit functions and loaded vector elements.

The *frequency* column of our example Huffman table, depicted in Table I, is not a property of the table itself but merely of the encoded bitstream. The analysis made in this section is based on the used data inputs, as described earlier. The lost energy, when a hit is found at the i-th table row, can be calculated using the previously defined $overhead$ function. Given a frequency distribution of the encoded bitstream and number of concurrent processing elements N, the actual lost energy for that bitstream can be calculated by summing the overhead per encoded symbol: $\sum_i frequency[i] \times overhead(i)$.

Performing this experiment for the used data input, based on VC-1 table 130, we observe the loss in energy as shown in Table II.

From Table II we can observe that for a 32-way implementation 79% of the loaded vector elements and applied hit

TABLE II
Energy loss of different N-way implementations

Type:	Energy loss:
reference	0%
8-way	62%
16-way	71%
32-way	79%

TABLE III
MEMORY AREA OVERHEAD.

Type:	Depth:	Bank width:	Banks:	Total:
reference	15,056	32 bits	1	$0.488mm^2$
reference packed	4,640	64 bits	1	$0.332mm^2$
8-way	608	128 bits	4	$0.710mm^2$
16-way	336	128 bits	8	$1.237mm^2$
32-way	256	128 bits	16	$2.349mm^2$

functions in the vector datapath is overhead, this occurs due to the speculative look-ahead. Increasing the N-way would result in a further loss of energy. The fact why this energy loss does not scale linearly comes from the frequency distribution, further along in the tail of the table the frequency is much lower. Due to this fact a low N-way has already a high energy penalty, further increasing the N-way yields lower additional energy penalties.

VII. AREA

In this section we calculate the memory area overhead. The overhead of the logic is not taken into account, it is too small with respect to the memory overhead. The area consumption for the memory introduced by this data parallel approach is calculated based on the TSMC 65nm general purpose process. The vector memory is build out of multiple SRAM banks each 128 bits wide, while the memory of the reference implementation is based on one bank of 32 bits wide. For wider memories the implementation with banks of 128 bits is the most efficient (currently we are unable to generate memories with a larger width). However an additional overhead is introduced by the usage of multiple banks, this occurs due to the row decoder and sense amplifiers in these banks. Furthermore, the depth of the banks is rounded to a multiple of 16 for efficiency.

Note that the area of the memories in Table III are calculated for VC-1, containing over 100 different Huffman tables. Clearly we have to pay a high price for high performance, by implementing this parallel approach, up to 2 mm^2 in 65nm. For standards requiring fewer Huffman tables, such as JPEG and MPEG-2, the required memory is much lower.

VIII. SPEED VS. ENERGY TRADE-OFF

From the previous sections it can be observed that there is a trade-off between area, speed and energy, which all scale by the used parallelism (N-way). This raises the question, how much a flexible N-way processor could reduce the consumed energy. For this we take the 32-way implementation and modify the first access. Changing all accesses would not result in any noticeable energy reduction since the changes are low that these elements are used. The first 32-way access is split-up in accesses of 2, 4, 16 or a single 32-way access, such that the sum of these accesses still equals 32. Only access patterns with increasing (or constant) N-ways are used, since these are the most beneficial due to the table distribution. We assume memory banks of 128-bit wide, each entry containing two Huffman entries (of 64-bit each). The initial number of cycles required for decoding a single Huffman entry (and updating the shifting buffer) is 9, every additional iteration in the Huffman lookup process has a penalty of 4 cycles. The number of cycles and consumed energy for many different access patterns is shown in Fig. 8. In this figure the energy required for accessing one memory bank is taken as one unit. Since our memory banks hold two vector elements, the 1-way implementation is not considered.

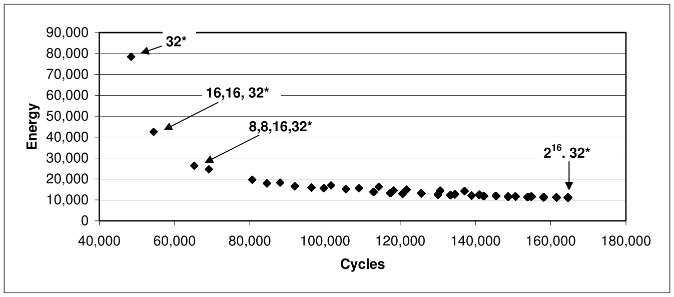

Fig. 8. Speed vs. energy trade off.

In Fig. 8 we indicated for four the access patterns. For example, '$8, 8, 16, 32*$' means two 8-way accesses, then a 16-way, followed by 32-way accesses for all remaining entries. The fastest, but most energy consuming, implementation is the 32-way implementation. The least energy consuming, but slowest, implementation is the 2-way implementation.

IX. CONCLUSION

In this paper a new technique is proposed that allows high speed Huffman decoding, while maintaining full flexibility of the Huffman tables and programmability of the processor. Furthermore there is no need to perform any modifications beforehand on the Huffman table. The SIMD approach taken in this paper results in high decoding speeds, high enough for real time decoding of modern video coding standards used for High Definition content. Moreover, we have shown that the usage of SIMD operations for Huffman decoding is beneficial.

It is shown that our reference linear search implementation has a decoding speed of 4.4 MBit/s, while the 32-way data parallel search has a speed of 106.0 MBit/s (with the processor running at 250 MHz). Hence we have a speed-up of approximately 24×, compared to our reference implementation.

However the usage of this data parallel approach, does come at a price. The memory area for the Huffman tables of the 32-way approach is 4.4× larger compared to our reference implementation. As shown in the experimental results section, gaps are introduced in the memory. When increasing the processors N-way, these table gaps grow larger, resulting in more overhead, but gaining in performance. Moreover, also energy is lost, ranging up to 79% for the 32-way implementation. Taking a slower approach by using accesses of 16-way, slightly reduces the speed but already results in a good energy reduction.

Finally, the proposed acceleration technique has been applied in a VC-1 decoder. It is capable of decoding advanced level 3 VC-1 encoded sequences, with peak bit-rates of 45 MBit/s.

References

[1] "Silicon Hive," http://www.siliconhive.com.

[2] W. Dally, P. Hanrahan, M. Erez, T. Knight, F. Labonté, J. Ahn, N. Jayasena, U. Kapasi, A. Das, J. Gummaraju *et al.*, "Merrimac: Supercomputing with Streams," *Proceedings of the ACM/IEEE SC2003 Conference*, 2003.

[3] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[4] P. SMPTE, "421M, VC-1 Compressed Video Bitstream Format and Decoding Process."

[5] S. Sudharsanan and M. Sinnathamby, "Support for Variable Length Decode on Embedded Processors," *Proceedings of the Workshop on Media and Signal Processors for Embedded Systems and SoCs*, pp. 33–40, 2004.

[6] J. Peng, X. Qin, J. Yang, X. Yan, and X. Chen, "A Programmable Bitstream Parser for Multiple Video Coding Standards," *Proceedings of the First International Conference on Innovative Computing, Information and Control-Volume 3*, pp. 609–612, 2006.

[7] Y. Chang, R. Chang, and L. Chen, "Design and implementation of a bitstream parsing coprocessor for MPEG-4 video system-on-chip solution," *International Symposium on VLSI Technology, Systems, and Applications*, pp. 188–191, 2001.

[8] M. Sima, E. Pol, J. van Eijndhoven, S. Cotofana, and S. Vassiliadis, "Entropy Decoding on TriMedia/CPU64," *Proceedings on System Architecture Modeling and Simulation Workshop*, 2002.

[9] M. Berekovic, H. Stolberg, M. Kulaczewski, P. Pirsch, H. Möller, H. Runge, J. Kneip, and B. Stabernack, "Instruction Set Extensions for MPEG-4 Video," *The Journal of VLSI Signal Processing*, vol. 23, no. 1, pp. 27–49, 1999.

[10] M. Sima, S. Cotofana, S. Vassiliadis, J. van Eijndhoven, and K. Vissers, "MPEG-compliant entropy decoding on FPGA-augmented TriMedia/CPU64," *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pp. 261–270, 2002.

[11] T.-H. Tsai and C.-N. Liu, "A low-latency multi-layer prefix grouping technique for parallel huffman decoding of multimedia standards," *Journal of Signal Processing Systems*, vol. 53, no. 3, pp. 323–333, 2008.

[12] R. Hashemian, "Memory efficient and high-speed search Huffman coding," *IEEE Transactions on Communications*, vol. 43, no. 10, pp. 2576–2581, 1995.

[13] T. Halfhill, "Silicon Hive Breaks Out," *Microprocessor Report, December*, vol. 1, 2003.