A Hybrid Implementation of Hamming Weight

Enric Morancho

*Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona Tech
Barcelona, Spain
Email: enricm@ac.upc.edu*

Abstract—The hamming weight (also known as population count) of a bitstring is the number of 1's in the bitstring. It has applications in scopes like cryptography, chemical informatics and information theory. Typical bitstring lengths range from the processor's word length to several thousands of bits.

A plethora of hamming weight algorithms have been proposed. While some implementations expose just scalar parallelism, others expose vector parallelism. Moreover, some implementations use special machine instructions that compute the hamming weight of a processor's word.

This paper presents a new hybrid scalar-vector hamming weight implementation that exposes both scalar and vector parallelism. This implementation will be useful on platforms that can exploit both kinds of parallelism simultaneously.

On a Sandy Bridge platform, our hybrid implementation outperforms by up to 1.23X and 1.6X the, to the best of our knowledge, best scalar and vector implementations respectively.

Keywords—hamming weight; population count; hybrid parallelism

I. INTRODUCTION

The hamming weight (also known as population count, sideways addition or bit counting) of a bitstring is the number of 1's in the bitstring. It has applications in scopes like coding theory [1], cryptography [2], chemical informatics [3] and chessplaying [4]. Typical bitstring lengths range from the processor's word length (32/64 bits) to thousands of bits (path-based fingerprints in chemical graphs).

Several algorithms have been proposed for computing the hamming weight of a word [5], [6], [7]. Moreover, since the early days of the computing era, some computer architects have defined specific machine instructions for computing it. The first processor to include such an instruction was, in the early 50's, the Mark II [8].

The trivial approach for computing the hamming weight of a multi-word bitstring consists in accumulating the hamming weight of each word of the bitstring. However, unless a specific machine instruction for computing hamming weight is available, it is not the best option from the performance point of view. The quest for high-performance hamming weight computing requires exposing and exploiting the available parallelism as much as possible. Proposed solutions expose either scalar parallelism or vector parallelism, but none exposes both parallelisms simultaneously.

Some current processors can dispatch both scalar and vector instructions simultaneously. However, existing hamming-weight implementations are not able to fully exploit the processor's dispatch width. We wonder if a hybrid scalar-vector implementation may largely exploit the dispatch width and, consequently, outperform existing implementations.

This work proposes a new hybrid implementation that exposes both scalar and vector parallelism simultaneously. On a Sandy Bridge platform, evaluations show that our proposal outperforms by up to 1.23 and 1.6 the, to the best of our knowledge, best scalar and vector implementations.

This paper is organized as follows. Section II describes the main algorithms for computing hamming-weight. Section III evaluates them and points out some relevant remarks. Section IV proposes the hybrid implementation, explores its design space and compares it with the best existing implementations. Finally, Section V concludes the paper.

II. ALGORITHMS FOR COMPUTING HAMMING WEIGHT

This section presents the main algorithms for computing the hamming weight of a bitstring. The granularity of some algorithms is the processor word; then, the hamming weight of a multi-word bitstring is obtained by accumulating the hamming weight of its words. Other algorithms accumulate the hamming weight of wider byte chunks.

A. Naïve

The trivial approach iterates across all the bits of a word and accumulates the bit values. However, this is the worst method in terms of efficiency because it does not exploit the intrinsic parallelism available in this computation. An optimization for sparsely-populated (or densely-populated) words consists in iterating just on the bits set to 1 (or to 0). Figure 1 shows these implementations for 32-bit words.

B. Memoization

This approach relies on defining a subword size, precomputing the hamming weights for all the possible subword values and keeping the precomputations in a lookup table. Then, the hamming weight of a word is computed by accumulating the hamming weights of its subwords. The lookup table has $2^{\text{bits per subword}}$ entries and each entry is, at least, $\lceil \log_2(\text{bits per subword} + 1) \rceil$ bits wide. To reduce

```

uint8_t
hw_naive(uint32_t w) {
    uint8_t i, cnt=0;

    for (i=0; i<32; i++, w = w>>1)
        cnt += w&0x1;
    return(cnt);
}

uint8_t hw_sparse(uint32_t w) {
    uint8_t cnt=0;

    for(; w; w=w&(w-1))
        cnt++;
    return(cnt);
}

uint8_t hw_dense(uint32_t w) {
    uint8_t cnt=32;

    for(; w!=0xFFFFFFFF; w=w|(w+1))
        cnt--;
    return(cnt);
}

```

Figure 1. Naïve implementations of hamming weight

the size of the lookup table, [9] proposed a technique that exploits regular patterns on the lookup-table contents.

The typical implementation of hamming weight based on memoization is scalar; Figure 2 shows this implementation for 8-bit subwords. However, a vector implementation is also possible. This approach, described in [10], implements memoization using a vector instruction introduced by the Supplemental Streaming SIMD Extensions 3 (SSSE3).

```

uint8_t T8[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, ..., 7, 8};

uint8_t
hw_memoization8(uint32_t w) {
    return(T8[w&0xFF] + T8[(w>>8)&0xFF] +
           T8[(w>>16)&0xFF] + T8[w>>24]);
}

```

Figure 2. Memoization: scalar implementation of hamming weight

SSSE3 offers the instruction `pshufb` (Packed Shuffle Bytes), a mighty vector instruction that shuffles the bytes of a vector register according to a mask recorded in another vector register. `pshufb` instruction has two 16-byte vector input operands. With the low-order nibble (4 bits) of each byte of the first operand, performs a 16-way parallel lookup on the second operand (interpreted as a 16-entry array) and retrieves sixteen low-order nibbles from the second operand.

This instruction can implement 4-bit subword memoization. Figure 3 shows an example of how this instruction converts sixteen nibbles into their sixteen hamming weights.

Figure 3. `pshufb` example computing hamming weight at nibble level

To compute the hamming weight of a 16-byte vector

register we must split it into its nibbles, perform two `pshufb`'s, and accumulate the results of each byte. The result is a vector register with sixteen 8-bit counters that contains the hamming weight of each byte. Figure 4 shows the core of the corresponding vector code. Note that, before using `pshufb`, the high-order bit of each byte of the first operand must be cleared because `pshufb` filters out the bytes with its high-order bit set.

As the hamming weight of each byte ranges from 0 to 8, an 8-bit counter can accumulate the hamming weight of up to $\lfloor \frac{255}{8} \rfloor$ bytes (31) without overflowing. Finally, the sixteen 8-bit counters must be accumulated into a wider counter.

```

__m128i w, wL, wH, t0;
__m128i ct; /* Set to __mm_setzero_si128() */
__m128i mk; /* Set to __mm_set1_epi8(0x80) */
__m128i T4; /* Set to __mmset_epi8(0, 1, 1, 2,
    1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4); */

...
w = __mm_load_si128(p);
wL = __mm_and_si128(mk, w); /*Low order nibble*/
wH = __mm_and_si128(mk, __mm_srli_epi16(w, 4)); /*High order n.*/
t0 = __mm_add_epi8(__mm_shuffle_epi8(T4, wL),
    __mm_shuffle_epi8(T4, wH));
ct = __mm_add_epi8(ct, t0);
...

```

Figure 4. Memoization: vector implementation of hamming weight

C. Parallel reduction at bit level

This approach, also known as SWAR (SIMD within a register) population count, performs a parallel reduction of the input word in $\lceil \log_2 \text{bits per word} \rceil$ levels. Table I shows an example of this computation over a byte.

Input	<table><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>								0	1	1	1	0	0	1	0
0	1	1	1	0	0	1	0									
Parallel reduction: level 1	<table><tr><td>01</td></tr></table>		01	<table><tr><td>10</td></tr></table>		10	<table><tr><td>00</td></tr></table>		00	<table><tr><td>01</td></tr></table>		01				
01																
10																
00																
01																
Parallel reduction: level 2	<table><tr><td>0011</td></tr></table>				0011	<table><tr><td>0001</td></tr></table>				0001						
0011																
0001																
Parallel reduction: level 3	<table><tr><td>00000100</td></tr></table>								00000100							
00000100																

Table I
EXAMPLE OF PARALLEL REDUCTION AT BIT LEVEL

Figure 5 depicts its 32-bit scalar implementation. The first sentence computes the hamming weight by 2-bit groups (level 1). Mask and shift operations face the appropriate bits and the scalar add instruction emulates sixteen 2-bit additions (note that this add instruction will never generate a carry propagation beyond 2-bit groups because the maximum result of each 2-bit addition is two). Analogously, the i -th sentence (level i) computes hamming weight by groups of 2^i bits and the add operation emulates 2^{5-i} 2^i -bit additions without generating carry propagation beyond groups.

This implementation can be simplified by using a bithack to compute the hamming weight of 2-bit groups and by minimizing the number of mask operations (Figure 6).

```

uint32_t
hw_parallel(uint32_t w) {
    w = (w & 0x55555555) + ((w>> 1) & 0x55555555); /*Level 1*/
    w = (w & 0x33333333) + ((w>> 2) & 0x33333333); /*Level 2*/
    w = (w & 0x0F0F0F0F) + ((w>> 4) & 0x0F0F0F0F); /*Level 3*/
    w = (w & 0x00FF00FF) + ((w>> 8) & 0x00FF00FF); /*Level 4*/
    w = (w & 0x0000FFFF) + ((w>>16) & 0x0000FFFF); /*Level 5*/
    return(w);
}

```

Figure 5. Parallel reduction: scalar implementation of hamming weight

```

uint32_t
hw_parallel(uint32_t w) {
    w = w - ((w>> 1) & 0x55555555); /* L. 1 */
    w = (w & 0x33333333) + ((w>> 2) & 0x33333333); /* L. 2 */
    w = (w + (w>> 4)) & 0x0F0F0F0F; /* L. 3 */
    w = w + (w>> 8); /* L. 4 */
    w = w + (w>> 16); /* L. 5 */
    return(w & 0x3F);
}

```

Figure 6. Simplified parallel reduction implementation of hamming weight

After analyzing the range of possible values at the bit-groups of each level, we point out that the widths of these emulated vector elements are not fully exploited. For instance, at level 1, 2 bits codify just 3 values; at level 2, 4 bits codify just 5 values,..., at level i , 2^i bits codify just $2^i + 1$ values. So, some data-width remains unused. Although this unused data-width can not be exploited in a single-word computation, we will exploit it in multi-word computations.

D. Merged parallel reduction

The main idea of the merged parallel reduction algorithm, also known as tree merging [11], is combining the intermediate results of several parallel reductions into a single intermediate result and then keep processing just the combined intermediate result. The implementation exploits the unused data width of the emulated SIMD registers.

Figure 7 shows a scalar implementation for the 3-word bitstring $w_a w_b w_c$. As the computation of level 1 on words w_a and w_b generates 16 2-bit groups with values 0, 1 or 2, we can accumulate another bit into each 2-bit group without overflowing: odd-numbered bit locations of w_c into w_a and even-numbered bit locations of w_c into w_b . Next, level 2 is computed just on w_a and w_b . As the value range for each 4-bit group is from 0 to 6, w_a and w_b can be merged without overflowing. Finally, parallel reduction continues just on w_a .

This approach can be extended to merge more partial parallel reductions. For instance [6] proposes a merging strategy at level 3 for 31 words. There exist scalar and vector implementations of this algorithm.

E. Bit slicing

This algorithm has been described in [11] and [4]. The key idea of this algorithm is transforming a $(2^n - 1)$ -word bitstring into n words, preserving indeed the hamming weight of the original bitstring. First, the transformation accumulates the i^{th} bit of all the words, that is, compacts

```

wa = (wa & 0x55555555) + ((wa>> 1) & 0x55555555); /*Level 1*/
wb = (wb & 0x55555555) + ((wb>> 1) & 0x55555555);
wa = wa + (wc & 0x55555555);
wb = wb + ((wc>>1) & 0x55555555);

wa = (wa & 0x33333333) + ((wa>> 2) & 0x33333333); /*Level 2*/
wb = (wb & 0x33333333) + ((wb>> 2) & 0x33333333);
wa = wa + wb;

wa = (wa & 0x0F0F0F0F) + ((wa>> 4) & 0x0F0F0F0F); /*Level 3*/
wa = (wa & 0x00FF00FF) + ((wa>> 8) & 0x00FF00FF); /*Level 4*/
wa = (wa & 0x0000FFFF) + ((wa>>16) & 0x0000FFFF); /*Level 5*/

```

Figure 7. Merged parallel reduction for three 32-bit words

$2^n - 1$ bits into a n -bit value. Next, gathers the j^{th} bit of all the accumulators into a word. Finally, the hamming weight of the original bitstring is equal to the sum of the hamming weights of the resulting words multiplied by the factor $2^{wordposition}$. Figure 8 depicts this process.

$$hw(w_0 w_1 \dots w_{2^n-2}) = \sum_{i=0}^{2^n-2} hw(w_i) = \sum_{j=0}^{n-1} 2^j \cdot hw(s_j)$$

Figure 8. Bit slicing algorithm: transformation scheme

The implementation relies on the parallel emulation of *bits_per_word* bit adders by using bit-wise logical instructions. For instance, for a 3-word bitstring ($n=2$), Figure 9 shows a scalar implementation of the transformation. Moreover, vector implementations are also possible.

```

s0 = w0 ^ w1 ^ w2;
s1 = (w0 & w1) | ((w0 ^ w1) & w2);

```

Figure 9. Bit slicing: scalar implementation of the transformation ($n=2$)

F. Processor support

Since the early days of computing era, several processors have been designed with a special machine instruction for computing the hamming weight of a word. For instance: Mark II (1954), IBM Stretch (1961), CDC 6600 (1964), Cray 1 (1976), Sun SPARCv9 (1995), Alpha 21264A (1999), IBM Power5 (2004) and ARM Cortex-A8 (2005).

In 2006 Intel announced the SSE4 instruction set. In addition to several new vector instructions, it also includes the `popcnt` instruction, a scalar instruction that computes

the hamming weight of a scalar register. SSE4 was split into SSE4.1 and SSE4.2 subsets; `popcnt` belongs to SSE4.2.

The first processors implementing SSE4.2 were released in 2007 (AMD K10 - Barcelona) and 2008 (Intel Core i7 - Nehalem). Table II shows the latency and the dispatch rate of `popcnt` instruction on AMD and Intel processors.

	AMD 15h		Intel Nehalem/Sandy Bridge
	32-bit	64-bit	32/64 bit
Latency (cycles)	4	6	3
Dispatch rate (inst/cyc)	1	0.25	1

Table II
TIMING CHARACTERISTICS OF `popcnt` INSTRUCTION ON AMD [12]
AND INTEL [13] PROCESSORS

III. EVALUATION OF EXISTING IMPLEMENTATIONS

This section describes our evaluation platforms, evaluates some implementations and points out some key remarks.

A. Evaluation environment

Table III describes our evaluation platforms (Nehalem and Sandy Bridge based). Although their DL1's cache size and associativity are the same, there are significant differences between these DL1s. While Nehalem's DL1 bandwidth is 16 bytes/cycle (load dispatch rate is one instruction per cycle), Sandy Bridge's is 32 bytes/cycle (load dispatch rate is two instructions per cycle). Also, several internal instruction buffers (reorder buffer, scheduler window,...) are larger in Sandy Bridge platform than in Nehalem platform.

	Intel Core i5-650	Intel Xeon E5-2630L
Microarchitecture	Nehalem	Sandy Bridge
Frequency (max turbo)	3.2(3.46) GHz	2(2.5) GHz
Cores	2	6
Reorder Buffer entries	128 μ -ops	168 μ -ops
Scheduler entries	36 μ -ops	54 μ -ops
Peak dispatch rate	6 μ -ops/cycle	
DL1	Size and assoc.	32KB, 8-way, 64Byte lines
	Bandwidth	128 bits/cycle
	In-flight loads	48
	Simult. misses	10
L2	256KB, 8-way, 64Byte lines	
L3	4MB, 16-way, 64B	15MB, 20-way, 64B

Table III
CHARACTERISTICS OF THE EVALUATION PLATFORMS

Table IV details the unithreaded implementations evaluated in this section. Programs have been compiled using 4.7.3 `gcc` compiler with optimization flags `-O3 -march=native` on a 64-bit Ubuntu distribution.

Our experiments consist in computing the hamming weight of several randomly initialized multi-word bitstrings (from bitstrings that fit in DL1 to bitstrings that do not fit in L3). For each bitstring length, we measure the bandwidth (in terms of Gibibytes per second) at which hamming weight is

Single-word wide implementations	
Naïve	<code>hw_naive</code> implementation (Figure 1)
Mem-8	Memoization with a 2^8 -entry lookup table
Mem-16	Memoization with a 2^{16} -entry lookup table
Par.Red.	Parallel reduction at bit level over 64-bit words
SSE4.2	Uses 64-bit scalar instruction <code>popcnt</code>

Multi-word wide implementations	
Merged	Scalar merged parallel reduction over 30 64-bit words at level 3
Merged-V	Vector merged parallel reduction over 30 128-bit words at level 3 (SSE2)
Slice	Scalar bit slicing over 7 64-bit words
Slice-V	Vector bit slicing over 7 128-bit words (SSE2)
Mem-4	Vector memoization with a 2^4 -entry lookup table (SSSE3 vector instruction <code>pshufb</code>)

Table IV
EVALUATED IMPLEMENTATIONS

computed. As the evaluated platforms implement Intel Turbo Boost technology (in some conditions, processors operate above their theoretical clock speed), to obtain repeatable measurements platforms are exclusively devoted to our evaluations, hyperthreading is disabled and OS is configured to permanently demand maximum processor performance.

Our evaluations consider two scenarios: uncached (no portion of the bitstring resides in DL1, L2 or L3 before starting the computation of the hamming weight) and cached (the bitstring already resides in the cache hierarchy when the computation begins). In the uncached scenario, we flush data caches before taking each measure.

B. Results

1) *Uncached bitstrings*: Figure 11-a shows the performance of the evaluated single-word wide implementations on the Nehalem platform. As expected, Naïve implementation performs worst due to the large number of machine instructions needed to process a single word. The performance of Mem-16 steadies only for large bitstrings. This is due to the impact of the compulsory cache misses produced accessing the 2^{16} -entry lookup table. We repeated the experiment loading the lookup table into cache before measuring performance; then we get steady performance for all evaluated bitstring lengths. Parallel reduction outperforms the previous implementations. Finally, SSE4.2 is the best option, due to both the short latency and the high dispatch rate of `popcnt` instruction (Table II).

Figure 11-b shows the performance of the multi-word wide implementations on the Nehalem platform. In steady state, they outperform all single-word wide algorithms but SSE4.2. Slice implementation performs worst. The remaining implementations behave similar up to bitstring lengths of 2^{22} bytes; then, while the performance of Merged implementations saturates, the performance of both Mem-4 and Slice-V show a performance increment. Finally, the performance

of SSE4.2 and Mem-4 implementations are almost the same.

For short bitstring lengths (up to 2^{15} bytes), the performance of SSE4.2 and multi-word implementations is highly dependent on the bitstring length. In steady state, the main-memory accesses to the bitstring are overlapped with hamming-weight computations; however, the main-memory accesses that load the initial words of the bitstring can not be overlapped with computations. Then, the shorter the workload, the larger the relative impact on performance of the non-overlapped memory accesses.

Beyond 2^{15} -byte bitstrings, the performance of some implementations is almost independent on the bitstring length. However, the performance of Slice-V, Mem-4 and SSE4.2 grows noticeably for bitstring lengths larger than 2^{22} bytes (Nehalem's L3 cache size). We consider that this effect can be due to hardware prefetching. As the evaluated implementations are sequentially accessing memory, hardware prefetchers can easily identify this pattern.

To verify this hypothesis, we perform some evaluations disabling hardware prefetching. As Nehalem platform does not allow to disable hardware prefetching, we perform the experiment on a Core2 platform with a 4MB L3 cache. Accessing the Machine Specific Register 1a0h, we disable the four hardware prefetchers of our Core2 processor. Figure 10 shows the impact of hardware prefetching on Mem-4 implementation (uncached scenario). For Mem-4 implementation, hardware prefetchers are able to increase performance up to 2.6X for bitstrings that do not fit in L3.

Figure 10. Impact of hardware prefetching on the performance of Mem-4 implementation on a Core2 platform

Figures 11-c and -d show performance results on Sandy Bridge platform. Main differences with respect to Nehalem's results are that the gap between Mem-4 and SSE4.2 implementations widens; also Mem-16 and Slice-V implementations slightly outperform Par.Red. and Mem-4 implementations respectively; finally, the effect of hardware prefetching is evenly distributed among all bitstring lengths.

2) *Cached bitstrings*: Figures 11-e, -f, -g and -h show the performance results on both platforms.

Performance of Naïve, Mem-8, Mem-16, Par.Red and Slice implementations is almost independent on the bitstring length. These implementations do not benefit from accessing data that is already cached because their bottleneck is not the memory latency; their performance is almost the same that in the uncached scenario (except for the shortest bitstrings).

The remaining implementations benefit from accessing cached data. In steady state, the speedup with respect to the uncached scenario is around 1.2 -Nehalem- and 1.11 -Sandy Bridge- (Merged implementation), 1.35/1.26 (Merged-V), 1.7/1.35 (Slice-V), 1.9/1.5 (Mem-4) and 2.7/1.6 (SSE4.2).

For bitstrings that fit in DL1, while the performance of SSE4.2 in Sandy Bridge almost steadies, its performance in Nehalem peaks only for bitstrings lengths equal to DL1 cache size. Probably, the learning phase of some microarchitectural enhancements (for instance, the loop stream detector) in Sandy Bridge is shorter than in Nehalem.

As `popcnt`'s dispatch rate is one instruction per cycle (Table II), peak performance of SSE4.2 implementation is 8 bytes/cycle, that is, 25.8 GiB/s -Nehalem- and 18.6 GiB/s -Sandy Bridge-. For bitstrings that fit in DL1, SSE4.2 implementation performs up to 90% of its peak performance.

Finally, the bitstring length at which performance reaches its minimum value depends on the platform: 2^{23} bytes in Nehalem, 2^{25} bytes in Sandy Bridge. This value is related to the L3 cache size of both platforms: 4MB and 15MB respectively (Table III). In all cases, for bitstrings larger than L3 cache, performance drops down until reaching the steady performance of the uncached scenario.

C. Conclusions

After analyzing these results we point out some conclusions:

- The scalar implementation SSE4.2 clearly outperforms the remaining implementations in all scenarios. The best vector implementations are Mem-4 and Slice-V.
- SSE4.2 implementation executes a loop that traverses memory sequentially: it loads a 64-bit word, executes `popcnt` instruction and accumulates the result. According to Table II, the dispatch rate of `popcnt` instruction is, at most, one instruction per cycle; consequently, hamming weight can be computed at a peak bandwidth of 8 bytes/cycle. However, Table III shows that the available DL1 bandwidth of our platforms is larger (16 and 32 bytes per cycle). Then, an implementation that relies only on `popcnt` instruction can not consume the available DL1 bandwidth.
- SSE4.2 implementation is fully scalar. However, the platforms used in this work can dispatch out-of-order up to six micro-ops per cycle: three ports are devoted for memory instructions and three ports can dispatch both scalar/vector instructions. Then SSE4.2 implementation can not fully exploit the three ports devoted to integer/vector instructions.

a) Nehalem, single-word, uncached

b) Nehalem, multi-word, uncached

c) Sandy Bridge, single-word, uncached

d) Sandy Bridge, multi-word, uncached

e) Nehalem, single-word, cached

f) Nehalem, multi-word, cached

g) Sandy Bridge, single-word, cached

h) Sandy Bridge, multi-word, cached

Figure 11. Performance of the evaluated implementations (Table IV) on both platforms

As SSE4.2 implementation is neither fully exploiting dispatch rate nor DL1 bandwidth, we wonder if SSE4.2 implementation can be outperformed by a hybrid implementation that makes use not only of the scalar instruction `popcnt` but also of vector instructions.

IV. PROPOSED HYBRID IMPLEMENTATION

A. Design

This subsection presents our hybrid implementation for computing the hamming weight of a bitstring.

We propose to combine both SSE4.2 (scalar) and Mem-4 (vector) implementations into a hybrid implementation. The idea is distributing the bitstring words between both the scalar functional units and the vector functional units. The challenge is distributing the input words in a balanced way among the execution units.

Our proposal iterates through the bitstring. Each loop iteration processes a fixed-sized chunk of the bitstring. Our proposal statically distributes the chunk between the scalar and the vector functional units. The design-space of this proposal is determined by two dimensions:

- the number of bytes of the chunk processed by the scalar functional units (S)
- the number of bytes of the chunk processed by the vector functional units (V)

Each configuration is characterized by the tuple (S,V), where $S + V$ is the number of bytes processed at each loop iteration, that is, the chunk length.

We have explored the design space of our hybrid implementation by evaluating configurations with a chunk length of up to 80 bytes, that is, (16,16), (32,16), (16,32), (48,16), (32,32), (16,48), (64,16), (48,32), (32,48) and (16, 64).

Figure 12 shows the results of the design-space exploration of our proposal for both scenarios (cached and uncached) on our two platforms. Each graph shows the relative performance with respect to SSE4.2 implementation (the higher, the better). As is not feasible to plot the individual results of all evaluated configurations, we present the performance range of the hybrid configurations on each bitstring length; the performance of all evaluated hybrid configurations lays in the grey area of each graph. Also, we plot the detailed results of just some relevant configurations.

To begin with, no hybrid configuration outperforms the remaining hybrid configurations for all evaluated bitstring lengths.

On Nehalem platform (Figures 12-a and -b) we observe that the hybrid implementations outperform SSE4.2 implementation up to 1.06X for bitstrings that do not fit L3. In the uncached scenario, we also observe that some hybrid configurations outperform SSE4.2 for bitstring lengths that fit in L2 or L3.

On Sandy Bridge platform (Figures 12-c and -d) results show a wider potential. On the uncached scenario, hybrid

configurations outperform SSE4.2 implementation by up to 1.11X for bitstrings that do not fit in DL1. On the cached scenario, the speed-up depends on the bitstring length: up to 1.19X for strings that fit in DL1 and up to 1.23X for strings that fit in L3. Finally, hybrid configuration (32,48) outperforms SSE4.2 for all evaluated bitstring lengths but 2^{10} bytes.

The performance differences between both platforms are due to several microarchitectural factors. The dispatch rate of `popcnt` instruction is one instruction per cycle in both platforms. However, while Nehalem's dispatch rate for load instructions is only one instruction per cycle (regardless the data width, 8 bytes or 16 bytes), Sandy Bridge's dispatch rate is two loads per cycle. Also, Sandy Bridge's instruction buffers are deeper than the corresponding buffers in Nehalem (Table III: reorder buffer, scheduler, in-flight loads). Deeper instruction buffers allow exposing more instruction level parallelism and help hiding the latency of DL1 misses.

We conclude that some hybrid configurations outperform SSE4.2 implementation. However, the performance potential of the hybrid implementation is bigger in Sandy Bridge than in Nehalem due to the microarchitectural features of both platforms (instruction-buffer sizes, DL1 bandwidth and dispatch rates). Although the best hybrid implementation depends on the bitstring length, we pick only one configuration for each platform: (32,32) for Nehalem and (32,48) for Sandy Bridge.

B. Evaluation of the proposed hybrid implementation

This subsection compares the performance of the (32,32) -Nehalem- and (32,48) -Sandy Bridge- hybrid configurations versus Mem-4 and SSE4.2 implementations.

Figures 13-a and -b show the performance of the selected implementations on Nehalem platform (uncached and cached scenario respectively). In the uncached scenario, for bitstrings larger than L2 (2^{18} bytes), the selected hybrid configuration outperforms SSE4.2 implementation by up to 1.07X. In the cached scenario, the selected hybrid configuration outperforms SSE4.2 just for the bitstrings larger than L3 by 1.04X.

Figures 13-c and -d show the performance of the selected implementations on Sandy Bridge platform (uncached and cached scenario respectively). On the uncached scenario, Hybrid implementation outperforms the other implementations for bitstrings larger than L2. For bitstring lengths larger than L2 but shorter than L3, the speedup with respect to SSE4.2 and Mem-4 is about 1.07X and 1.26X respectively. On Sandy Bridge platform we observe that, on the cached scenario, (32,48) hybrid configuration outperforms the other implementations. Speedup with respect to SSE4.2 implementation is up to 1.15X, 1.18X and 1.22X for bitstrings that fit DL1, L2 and L3 respectively. Speedup with respect to Mem-4 implementation is even larger, about 1.6X in all cases. For large bitstrings that do not fit L3, speedup is

Figure 12. Design-space exploration of the hybrid proposal (the performance of all evaluated hybrid configurations lays in the grey area of each graph)

about 1.1X and 1.34X with respect to SSE4.2 and Mem-4 respectively.

Table V summarizes the speed-up achieved by our proposal with respect to SSE4.2 implementation on Sandy Bridge platform (cached scenario). According to the bitstring length (up to DL1/L2/L3 size and larger than L3), the table shows the potential speed-up revealed by the design space exploration and the speed-up achieved by the (32,48) configuration.

	Bitstring length			
	up to DL1	up to L2	up to L3	>L3
Potential speed-up	1.19	1.18	1.23	1.11
(32,48) speed-up	1.15	1.18	1.22	1.10

Table V

SPEED-UP OF OUR PROPOSAL WITH RESPECT TO SSE4.2 IMPLEMENTATION ON SANDY BRIDGE PLATFORM (CACHED SCENARIO)

V. CONCLUSION

This work has analyzed the problem of computing the hamming weight of a bitstring. After reviewing and evaluating the existing implementations, we have noticed that existing implementations expose either scalar parallelism or

vector parallelism. We propose a new hybrid implementation that exposes both kinds of parallelism simultaneously. This implementation is useful in platforms that can exploit both kinds of parallelism simultaneously.

Our evaluations on a Sandy Bridge platform show that this proposal outperforms the, to the best of our knowledge, best existing scalar and vector implementations. Our design-space exploration reveals that the speed-up of the evaluated hybrid implementations with respect to the best existing scalar implementation is up to 1.19X, 1.18X and 1.23X depending on which cache level holds the bitstring (DL1, L2 and L3 respectively). For larger bitstrings, the speedup is 1.11X. Focusing just on one configuration of the hybrid implementation, (32,48) configuration outperforms SSE4.2 implementation by 1.15X, 1.18X and 1.22X for bitstrings that fit DL1, L2 and L3 respectively; and by 1.1X for bitstrings that do not fit in L3. The speedup with respect to the Mem-4 vector implementation is around 1.6X.

Our future work includes extending this analysis to the platform Intel Haswell because it introduces a new set of 256-bit integer vector instructions (AVX2) that increases the potential of our hybrid approach.

a) Nehalem, uncached

c) Sandy Bridge, uncached

b) Nehalem, cached

d) Sandy Bridge, cached

Figure 13. Performance of SSE4.2, Mem-4 and selected configurations of Hybrid implementation on evaluated platforms

ACKNOWLEDGMENT

We thankfully acknowledge the support of the Spanish Ministry of Education (TIN2012-34557) and the European Commission in the context of the HiPEAC3 Network of Excellence (FP7/ICT 287759).

REFERENCES

- [1] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.
- [2] M. Gomulkiewicz and M. Kutyłowski, "Hamming weight attacks on cryptographic hardware - breaking masking defense," in *Proceedings of the 7th European Symposium on Research in Computer Security*, ser. ESORICS '02. London, UK, UK: Springer-Verlag, 2002, pp. 90–103. [Online]. Available: <http://dl.acm.org/citation.cfm?id=646649.699366>
- [3] I. S. Haque, V. S. Pande, and W. P. Walters, "Anatomy of high-performance 2d similarity calculations," *Journal of Chemical Information and Modeling*, vol. 51, no. 9, pp. 2345–2351, 2011. [Online]. Available: <http://pubs.acs.org/doi/abs/10.1021/ci200235e>
- [4] Chess Programming Wiki. (2007) Population count. [Online]. Available: <http://chessprogramming.wikispaces.com/Population+Count>
- [5] M. Beeler, R. W. Gosper, and R. Schroeppel, "Hakmem," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1972.
- [6] H. S. Warren, *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [7] A. Oram and G. Wilson, *Beautiful code*, 1st ed. O'Reilly, 2007.
- [8] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, 12th ed. Addison-Wesley Professional, 2009.
- [9] E. El-Qawasmeh and W. Al-Qarqaz, "Reducing lookup table size used for bit-counting algorithm," in *Proceedings of the 4th international multiconference on Computer Science and Information Technology (CSIT2006)*, Petra, Jordan, Jan. 2006, pp. 524–531.
- [10] W. Muła. (2008) SSSE3: fast popcount. [Online]. Available: <http://wm.ite.pl/articles/sse-popcount.html>
- [11] C. Lauradoux. (2009) Hamming weight. [Online]. Available: <http://planete.inrialpes.fr/~lauradoux/>
- [12] *Software Optimization Guide for AMD Family 15h Processors (Rev. 3.06)*, AMD, 2012.
- [13] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel, 2013.