TechRxiv
Powered by IEEE

A Depth-First Iterative Algorithm for the Conjugate Pair Fast Fourier Transform

This paper was downloaded from TechRxiv (https://www.techrxiv.org).

LICENSE

CC BY-NC-SA 4.0

SUBMISSION DATE / POSTED DATE

26-12-2020 / 29-12-2020

CITATION

Becoulet, Alexandre; Verguet, Amandine (2020): A Depth-First Iterative Algorithm for the Conjugate Pair Fast Fourier Transform. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.13489392.v1

DOI

10.36227/techrxiv.13489392.v1

A Depth-First Iterative Algorithm for the Conjugate Pair Fast Fourier Transform

Alexandre Becoulet, *Freebox* and Amandine Verguet, *Université Paris-Sud*

***Abstract*—The Split-Radix Fast Fourier Transform has the same low arithmetic complexity as the related Conjugate Pair Fast Fourier Transform. Both transforms have an irregular datapath structure which is straightforwardly expressed only in recursive forms. Furthermore, the conjugate pair variant has a complicated input indexing pattern which requires existing iterative implementations to rely on precomputed tables. It however allows optimization of the memory bandwidth as it requires a single twiddle factor load per radix-4 butterfly. In existing algorithms, this comes at the cost of using additional precomputed tables or performing recursive function calls. In this paper we present two novel approaches that handle both the butterfly scheduling and the input index generation of the Conjugate Pair Fast Fourier Transform. The proposed algorithm is cache-friendly because it is depth-first, non-recursive and does not rely on precomputed index tables. In order to achieve this, we relate the butterfly execution pattern of the Split-Radix and Conjugate Pair FFTs to the *binary carry sequence*. Based on this finding, we describe how common integer arithmetic and bitwise operations can be used to perform input reordering and depth-first traversal of the transform datapath with $\mathcal{O}(1)$ space complexity.**

***Index Terms*—FFT, Split-Radix, Conjugate Pair, Fast Fourier Transform, Binary Carry Sequence**

I. Introduction

THE performance of fast Fourier transform (FFT) software and hardware implementations depends on many factors, including algorithmic complexity, arithmetic complexity and memory access patterns. It is challenging to keep all aspects under control as certain design choices may lead to trade-offs. Moreover, high performance implementations use a myriad of optimizations that undermine simplicity [1].

Although both the Split-Radix Fast Fourier Transform (SRFFT) [2], [3] and the more common Cooley-Tukey FFT (CTFFT) [4] algorithms share the same algorithmic complexity of $\mathcal{O}(N \log N)$, the former has a lower arithmetic operation count than the latter. Let N be a power of 2. CTFFT has a regular datapath structure, forming a discrete Fourier transform (DFT) of length N from two DFTs of length $N/2$, assuming that radix-2 butterflies are used. The SRFFT algorithm uses L-shaped butterflies, forming a DFT of length N from one DFT of length $N/2$ and two DFTs of length $N/4$. The lower arithmetic complexity of SRFFT allows a performance improvement at the cost of more complex implementations due to the irregular datapath structure [5]–[7].

*A. Becoulet is with Freebox SAS, PARIS 75008, FRANCE. e-mail: alexandre.becoulet@free.fr

A. Verguet was with Université Paris-Sud, Université Paris-Saclay, ORSAY 91405, FRANCE.

This work has been submitted on 11-Apr-2020 to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

function CP_BF2($out[\,], h, in[\,], i_0, i_1$)
  $out[h] \leftarrow in[i_0] + in[i_1]$
  $out[h+1] \leftarrow in[i_0] - in[i_1]$

function CP_BF4^S($l, out[\,], h, k$)
  $a \leftarrow out[h+k]$
  $b \leftarrow out[h+k+l/4]$
  $c \leftarrow out[h+k+l/2]$
  $d \leftarrow out[h+k+3l/4]$
  $\omega \leftarrow e^{-2\mathrm{i}\pi k/l}$
  if $S = 1$ **then**
    $out[h+k] \leftarrow a + (\omega c + \overline{\omega} d)$
    $out[h+k+l/4] \leftarrow b - \mathrm{i}\,(\omega c - \overline{\omega} d)$
    $out[h+k+l/2] \leftarrow a - (\omega c + \overline{\omega} d)$
    $out[h+k+3l/4] \leftarrow b + \mathrm{i}\,(\omega c - \overline{\omega} d)$
  else if $S = -1$ **then**
    $out[h+k] \leftarrow a + (\overline{\omega} c + \omega d)$
    $out[h+k+l/4] \leftarrow b + \mathrm{i}\,(\overline{\omega} c - \omega d)$
    $out[h+k+l/2] \leftarrow a - (\overline{\omega} c + \omega d)$
    $out[h+k+3l/4] \leftarrow b - \mathrm{i}\,(\overline{\omega} c - \omega d)$

Fig. 1. These functions implement the radix-2 and radix-4 butterflies of CPFFT. Together, they provide transforms of fixed lengths 2 and 4, used as base cases in the algorithm. The radix-4 butterfly is where the forward transform differs from the inverse transform. Unlike other FFT algorithms, a single twiddle factor ω is required in the radix-4 butterfly of CPFFT.

The Conjugate Pair FFT (CPFFT) algorithm has originally been introduced as a variant of the SRFFT algorithm that further reduces the number of arithmetic operations [8] while retaining the same datapath structure. However, the claim on lower arithmetic complexity was later disproved [9], [10] and interest for the algorithm declined. Interestingly, CPFFT requires less memory bandwidth than the original SRFFT. This led to a recent regain of interest for the algorithm due to the optimization challenges posed by hierarchical memory architectures used in modern computers. In the design of CPFFT, Kamar and Elcherif used a different subset of the samples in the second sub-transform of length $N/4$. This allows using $e^{-2\mathrm{i}\pi k/l}$ and $e^{2\mathrm{i}\pi k/l}$ as twiddle factors instead of $e^{-2\mathrm{i}\pi k/l}$ and $e^{-6\mathrm{i}\pi k/l}$. CPFFT thus uses a pair of conjugates as twiddle factors where SRFFT uses two unrelated values. This makes CPFFT require only a single complex root of unity load per radix-4 butterfly, instead of two (Fig.1). Moreover, two butterflies in the same block can share the same twiddle factor, further halving the required number of twiddle-related

memory loads [11]. This design however makes the CPFFT algorithms more complicated because it requires taking some of the samples backward in time.

Indeed, in addition to the irregular datapath structure inherited from SRFFT, CPFFT has a complicated indexing pattern of the input array. When decimation in frequency (DIF) is used instead of decimation in time (DIT), the reordering takes place at the output [5]. The indices of the second sub-transform of length $N/4$ are cyclically offset by a negative amount [11], [12]. The proper reordering is however naturally handled by the recursive form of the algorithm.

Incontestably, the SRFFT and CPFFT algorithms are simple when expressed in a recursive form (Fig.2). In the past, efforts have been made to devise iterative algorithms for SRFFT, resulting in algorithms that perform a breadth-first traversal of the transform datapath [6], [7].

Depth-first traversal of the FFT datapath is favored in recent research as it is more memory cache-friendly than the breadth-first approach [1]. Even though the function call overhead may impact the performance of recursive algorithms [13], they are guaranteed to perform a depth-first traversal of the datapath. The memory access pattern of the depth-first approach improves cache usage because it tends to work on the same samples as much as possible, iterating over stages before moving to another part of the data array [14]. Algorithms relying on the function call stack as well as algorithms relying on an explicit stack data structure to implement the depth-first order are all considered recursive by the authors.

In this paper, we describe a depth-first and non-recursive algorithm that computes the DIT CPFFT using two novel approaches which support scheduling of butterflies and generation of input indices. Both tasks are performed without relying on precomputed tables. Moreover, the novel iterative approach does not require a stack, allowing the depth-first traversal of the transform datapath to yield a space complexity of $\mathcal{O}(1)$ instead of $\mathcal{O}(log(N))$, circumventing the associated memory accesses altogether. This is the main contribution of the paper. The reduced number of twiddle-related memory loads common to all CPFFT variants further lowers the required memory bandwidth. All these properties render the algorithm optimized in terms of memory access count and locality. Moreover, the cache-obliviousness [15] of the recursive CPFFT algorithm is retained.

Because the proposed algorithm computes the conventional CPFFT, some of its properties are shared with the previously known CPFFT algorithms. Indeed, we can see that the recursive algorithm presented in Fig.2 and the proposed algorithm presented in Fig.6 both rely on the butterfly provided in Fig.1 that is derived from the original CPFFT paper [8]. As a result, the number of arithmetic operations is as detailed in [10] and the floating-point accuracy is expected to be as given in [11].

The proposed algorithm has a concise and regular structure. It is composed of three nested loops that iterate, from the outermost to the innermost loop, over the data arrays, the transform stages and the blocks of butterflies. In addition to the three loops, a single conditional statement is needed in order to implement its entire control flow. All computations related to the datapath traversal and index generation processes can be implemented using integer arithmetic and bitwise operations available as general-purpose instructions in modern processors. The simple control flow and use of low gate count operators is expected to allow straightforward implementations of the algorithm in hardware as well.

function CPFFT_REC$^S_N(l, out[N], h, in[N], i)$
  $i_0 \leftarrow i \bmod N$
  if $l = 1$ **then**
    $out[h] \leftarrow in[i_0]$
  else if $l = 2$ **then**
    $i_1 \leftarrow (i + N/l) \bmod N$
    CP_BF2(out, h, in, i_0, i_1)
  else
    CPFFT_REC$^S_N(l/2, out, h, in, i)$
    CPFFT_REC$^S_N(l/4, out, h + 2l/4, in, i + N/l)$
    CPFFT_REC$^S_N(l/4, out, h + 3l/4, in, i - N/l)$
    for $b \leftarrow 0$ **to** $l/4 - 1$ **do**
      CP_BF4$^S(l, out, h, b)$

function DFT$_N(out[N], in[N])$
  CPFFT_REC$^1_N(N, out, 0, in, 0)$

function DFT_INVERSE$_N(out[N], in[N])$
  CPFFT_REC$^{-1}_N(N, out, 0, in, 0)$

Fig. 2. The recursive CPFFT algorithm has the same structure as the SRFFT algorithm: A DFT of length N is fromed by a single DFT of length $N/2$ and two DFTs of length $N/4$, yielding a total of three recursive function calls. This algorithm is described in [12].

In the next section, we give a brief review of the published algorithms that share some properties with the proposed algorithm. In the subsequent sections, we detail how the proposed iterative algorithm schedules execution of the butterflies, yielding the same depth-first order as the recursive algorithm. We then explain how this process can be extended in order to generate indices suitable for accessing the input array of CPFFT in appropriate order. In the last section, we present several experiments that compare the performance of the proposed algorithm with many FFT algorithm variants.

II. Related work

In 1986, Sorensen, Heideman, and Burrus proposed an SRFFT algorithm [7]. They managed to perform traversal of the irregular datapath of SRFFT using an iterative algorithm. The algorithm is composed of three nested loops. Because breadth-first order is used, the outer loop iterates over stages. The second loop iterates over adjacent radix-4 butterflies and the innermost loop handles butterflies sharing the same twiddles in the stage. Two separate loops are used to reorder the input and to process the first stage composed of radix-2 butterflies.

In 1992, Skodras and Constantinides noticed that redundancies exist in the butterfly scheduling part of the Sorensen SRFFT algorithm. They proposed a modified algorithm that relies on two precomputed tables [6]. The outer loop still makes the algorithm breadth-first. The lookup tables contain

Algorithm	Sorensen [7]	Skodras [6]	Johnson [11]	Blake [12]	Blake [16]	Lin [17]	Ocovaj [13]	Proposed
Traversal	Breadth-First	Breadth-First	Depth-First	Depth-First	Hybrid	Unspecified	Breadth-First	Depth-First
Structure	Iterative	Iterative	Recursive	Recursive	Hybrid	Unspecified	Iterative	Iterative
Butterfly	SRFFT	SRFFT	Novel	CPFFT	CPFFT	Novel	CPFFT	CPFFT
Twiddles grouping	By stage	By stage	By block	By block	By block	Unspecified	By block	By block
Precomputed indices	No	Yes	No	No	Yes	Yes	Yes	No

Fig. 3. A summary of the properties of algorithms related to CPFFT described in the literature. We can see that association of the depth-first traversal approach and the iterative structure does not exist in published algorithms.

the indices and number of radix-4 butterflies sharing common twiddles in a stage and are thus used in the innermost loop.

In 2007, Johnson and Frigo proposed a CPFFT variant with a reduced number of arithmetic operations [11]. The authors notice that CPFFT allows to share a twiddle between two butterflies. They provide a recursive algorithm that computes their new FFT.

In 2013, Blake, Witten and Cree proposed the *FFTS* software package that relies on CPFFT [16]. The proposed algorithm uses two different approaches for the first and subsequent stages of the transform. The first stage of the transform is handled by an iterative pass which also fetches input samples in the right order. A precomputed table of input indices is used for that purpose. The next stages are then processed in depth-first order by the recursive CPFFT algorithm. The implementation uses SIMD parallelism and run-time specialized codelets for base case processing. This design allowed *FFTS* to outperform state-of-the-art FFT implementations [18].

Using specialized codelets for small sub-transforms, i.e. hard-coded transforms for base cases, is a common practice in high performance FFT implementations [1]. Even if it is sometimes considered an implementation detail that does not always appear in the algorithm description, hard-coded base cases improve performance by not forcing data to be stored into memory between stages of smallest sub-transforms.

The same year, Lin and Chung proposed a variant of CPFFT with a more regular datapath structure than the original SRFFT [17]. The algorithm however still yields an unconventional ordering of output values due to the use of conjugate twiddle pairs. The authors propose a separate iterative algorithm that computes a single output index. Since finding all indices requires executing this additional algorithm N times per transform, we assume that a practical way to implement this FFT would be to rely on a precomputed table of indices. The paper focuses on butterfly and datapath designs and does not suggest a butterfly scheduling strategy.

In 2014, Zheng and Li proposed three novel DFTs [19] based on [11]. The paper focuses on the arithmetic complexity of their butterfly designs. It briefly discusses the twiddle factor access count of their approach as well as the input samples reordering in CPFFT. No algorithm structure is proposed.

The same year, Ocovaj and Lukac proposed a CPFFT algorithm for DSP platforms with SIMD instructions [13]. Their algorithm is not recursive in order to avoid function call overhead. Instead, they devise a breadth-first algorithm with a datapath traversal approach similar to that of the Skodras algorithm [6]. The paper focuses on SIMD-based optimization and use of precomputed tables for butterfly scheduling. Reordering of samples specific to CPFFT is however not covered.

As exposed in (Fig.3), we were unable to find a published non-recursive algorithm that schedules the SRFFT butterflies in the cache-friendly depth-first order. Likewise, we found no existing iterative CPFFT algorithm that computes input/output indices on the fly, despite this being performed by the concise recursive algorithm (Fig.2).

III. Datapath traversal and butterfly scheduling

The structure of SRFFT and CPFFT is often represented using a tessellation of L-shaped butterflies. We choose to represent the datapath of CPFFT of length 32 with a clear boundary between processing stages (Fig.4), in order to better expose how the butterflies can be scheduled.

After every stage S_j, intermediate values are stored into the output array. At stage S_0, input values are either processed by radix-2 butterflies or copied directly to the output array. Stages S_1 to $S_{\log 2(N)-1}$ are processed in-place on the output array, conditionally computing blocks of radix-4 butterflies. This representation shows that when walking through the output array for a given stage, blocks of butterflies are either computed or skipped following the non-periodic pattern $T_0 = (1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1...)$ for any value of N. Interestingly, this is an integer sequence known as the *parity of 2-adic valuation of n* [20] that has not been related to the SRFFT yet.

This sequence can be used by our algorithm to schedule radix-2 butterflies as appropriate for stage S_0 of the datapath. The depth-first traversal implies that the outer loop of the algorithm has to walk through the output array while the second nested loop walks through the transform stages. This second loop has to conditionally trigger computation of butterfly blocks while iterating over the transform stages.

We observe that for stage S_0, butterflies of radix-2 are only computed when the index of the output pair is associated to a 1 in the sequence T_0. In this paper, we assume that the first term of any sequence has index 0. For the higher stages $S_j|j > 0$, the same observation can be made regarding blocks of butterflies, with Fig.4 showing a 2^j scale factor of the pattern. For stage S_0, it is trivial to use the sequence T_0 to conditionally execute the radix-2 butterflies. For higher stages however, a scaled variant of the pattern T_0 cannot be used because we have to reach the output index of the last butterfly in the block before computing the whole block at once. This yields more sparse butterfly execution patterns for higher stages. For instances, the pattern for stage S_1 is

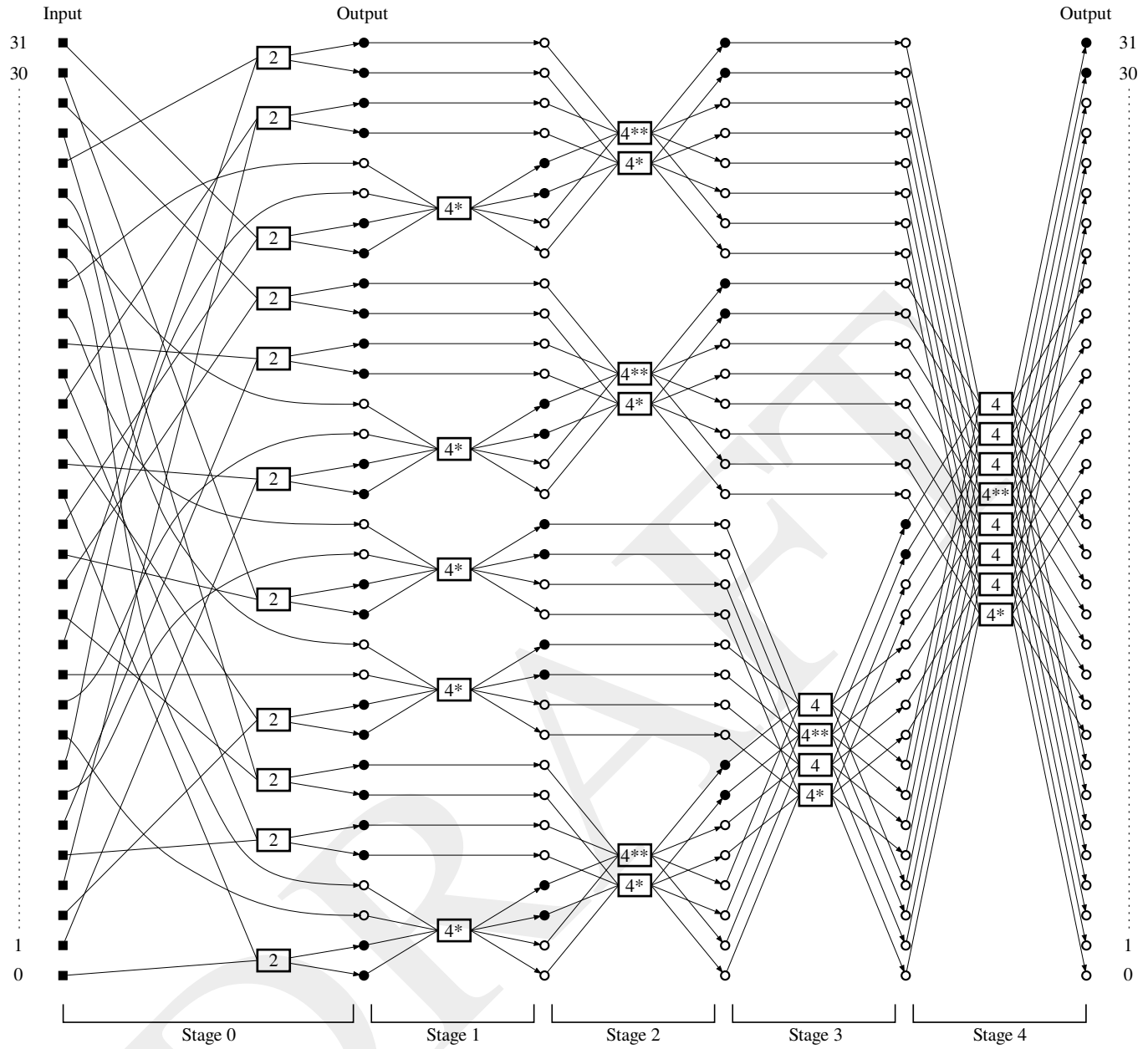

Fig. 4. Datapath of CPFFT of length 32. The input array is partially processed by radix-2 butterflies. Intermediate results are stored in the output array where radix-4 butterflies operate in-place and are grouped in blocks. When the depth-first order is used, a block is processed at once, as soon as the last pair of intermediate results becomes available from the previous stage. Block execution index is marked by a pair of filled circles at its output. The patterns formed by pairs of circles at the output of stage j is related to the T_j sequences. Radix-4 butterflies denoted by 4^* and 4^{**} use twiddles $\omega = e^0$ and $\omega = e^{-\mathrm{i}\pi/4}$ respectively that allow arithmetic simplifications.

$T_1 = (0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0...)$. Fortunately, there is an integer sequence that can be used to derive all T_j sequences. This is the *2-adic valuation of n*, also known as the *binary carry sequence* [21], defined as $C = (0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, 0, ...)$. The sequence T_0 is the parity of C.

C can be used to drive the whole datapath execution of SRFFT and CPFFT in depth-first order. Let h be the index in the output array and g the outer loop variable in our algorithm. In order to decide which butterflies must be computed at a given output pair index $g = \lfloor h/2 \rfloor$, the value $C(g)$ is used as follows: When $C(g)$ is odd, blocks of radix-4 butterflies are computed for odd stages S_1 to $S_{C(g)}$. When $C(g)$ is even, a radix-2 butterfly is computed for S_0, then blocks of radix-4 butterflies are computed for even stages S_2 to $S_{C(g)}$, unless $C(g) = 0$. This approach makes the second loop iterate over radix-4 stages that need processing, skipping other stages altogether.

In order to complete the butterfly scheduling part of our algorithm without relying on a precomputed table, we need an efficient way to compute the *binary carry sequence* C. The computation of $C(g)$ can be performed by counting the number of trailing zero digits of $g+1$ in binary representation [21]. This is a cheap operation in hardware. Software imple-

mentations may rely on the standard POSIX function `ffs` for that purpose. Some processors have a dedicated instruction to support this, as is the case on the *x86* architecture. However most processors only provide an instruction able to find the number of leading zero bits of an integer, which is related to computing the rounded down base 2 logarithm over $\mathbb{N}$. Thus, we propose a novel formula which takes advantage of the more common bit counting instruction:

$$C(g) = \lfloor \log_2(g \oplus (g+1)) \rfloor \tag{1}$$

It is manifest that the number of trailing zero bits of an integer corresponds to the most significant bit position where the carry propagates when subtracting 1. By using an exclusive or operation, we can extract the bitwise difference between two contiguous integers, yielding zeros for the most significant bits that were not impacted by the carry propagation and ones for the remaining trailing bits. Counting the number of leading zeros in this result is thus similar to counting the number of trailing ones. That is why either method can be used to compute the *binary carry sequence*. The first method is used in (1). This formula can thus be computed thanks to an *eXclusive OR* and a *Count Leading Zeros* or equivalent general-purpose instructions available on *x86*, *ARM*, *MIPS*, *PowerPC* and many other processor architectures.

Relying on a precomputed table to store the values of $C(g)$ used in our algorithm would require $N/2$ entries and is against our general strategy of reducing the number of memory accesses. Software experiments have shown that the bitwise method and the precomputed table method are both very fast when compared to the execution time of a CPFFT butterfly. Because of this, the raw performance of the method retained to compute $C(g)$ in the algorithm has no direct impact on the overall performance. However, we have noticed that the precomputed table method degrades the performance on large transforms due to cache pollution.

Because the two algorithms share the same datapath structure, the butterfly scheduling approach we have just described for CPFFT also works for SRFFT.

IV. Generation of indices for the input array

The input indexing of CPFFT looks irregular (Fig.4) and makes the datapath of stage S_0 specific to this algorithm. The sequence of indices used to access the input array of the transform is a distinctive permutation of the output indices. For instance, the sequence suitable for a transform of length 32 is defined as $I_{32} = (0, 16, 8, 24, 4, 20, 28, 12, 2, 18, 10, 26, 30, 14, 6, 22, 1, 17, 9, 25, 5, 21, 29, 13, 31, 15, 7, 23, 3, 19, 27, 11)$. It has no symmetry. Permutations for different transform lengths are not related, that is I_{N_0} is not a subsequence of I_{N_1} when $N_0 < N_1$.

The recursive implementation of CPFFT (Fig.2) composes the input indices as sums of powers of 2, negated powers of 2 or a mix thereof. When decomposed and ordered in output array order, the terms of the sums show interesting properties (Fig.5). First, the terms with greater magnitudes toggle with the lower binary digits of h. This property is shared with other FFT algorithms working in-place on the output array and is

$$\begin{array}{rrcccccc}
I_{32}(0) = & 0 & & & & & & \\
I_{32}(1) = & 16 & \equiv & +16 & & & & \\
I_{32}(2) = & 8 & \equiv & & +8 & & & \\
I_{32}(3) = & 24 & \equiv & & -8 & & & \\
I_{32}(4) = & 4 & \equiv & & & +4 & & \\
I_{32}(5) = & 20 & \equiv & +16 & & +4 & & \\
I_{32}(6) = & 28 & \equiv & & & -4 & & \\
I_{32}(7) = & 12 & \equiv & +16 & & -4 & & \\
I_{32}(8) = & 2 & \equiv & & & & +2 & \\
I_{32}(9) = & 18 & \equiv & +16 & & & +2 & \\
I_{32}(10) = & 10 & \equiv & & +8 & & +2 & \\
I_{32}(11) = & 26 & \equiv & & -8 & & +2 & \\
I_{32}(12) = & 30 & \equiv & & & & -2 & \\
I_{32}(13) = & 14 & \equiv & +16 & & & -2 & \\
I_{32}(14) = & 6 & \equiv & & +8 & & -2 & \\
I_{32}(15) = & 22 & \equiv & & -8 & & -2 & \\
I_{32}(16) = & 1 & \equiv & & & & & +1 \\
I_{32}(17) = & 17 & \equiv & +16 & & & & +1 \\
I_{32}(18) = & 9 & \equiv & & +8 & & & +1 \\
I_{32}(19) = & 25 & \equiv & & -8 & & & +1 \\
I_{32}(20) = & 5 & \equiv & & & +4 & & +1 \\
I_{32}(21) = & 21 & \equiv & +16 & & +4 & & +1 \\
I_{32}(22) = & 29 & \equiv & & & -4 & & +1 \\
I_{32}(23) = & 13 & \equiv & +16 & & -4 & & +1 \\
I_{32}(24) = & 31 & \equiv & & & & & -1 \\
I_{32}(25) = & 15 & \equiv & +16 & & & & -1 \\
I_{32}(26) = & 7 & \equiv & & +8 & & & -1 \\
I_{32}(27) = & 23 & \equiv & & -8 & & & -1 \\
I_{32}(28) = & 3 & \equiv & & & +4 & & -1 \\
I_{32}(29) = & 19 & \equiv & +16 & & +4 & & -1 \\
I_{32}(30) = & 27 & \equiv & & & -4 & & -1 \\
I_{32}(31) = & 11 & \equiv & +16 & & -4 & & -1
\end{array}$$

Fig. 5. Decomposition of input indices of a 32 point CPFFT as sums of powers of 2 and negated powers of 2, modulo N.

commonly handled by the bit-reversal operation [22]. Then, some of the sums are negative, which makes the input indices wrap modulo N. This is due to the third sub-transform of CPFFT using points taken backward in time [8]. Finally, terms can be arranged in a way that yields a pattern derived from the T_0 sequence, due to the specific datapath structure of SRFFT, as exposed in the previous section. When combined, these properties explain the apparently complicated permutation of indices used to fetch the input samples in DIT CPFFT.

Obviously, the I_N sequence is suitable for use in the outer loop of our algorithm which iterates over the output array. We therefore need an efficient way to compute this sequence. Let h_n be the binary digits of h:

$$\begin{cases} N = 2^L | L \in \mathbb{N} \\ h = \sum_{n=0}^{L-1} h_n 2^n \\ \forall n \notin \{0, ..., L-1\}\, h_n = 0 \end{cases} \tag{2}$$

Based on (2) and the three properties mentioned above, the sequence of interest can then be defined as:

$$I_N(h) = \left[\sum_{n=0}^{L-1} 2^n (-1)^{2-h_{L-2-n}} \left[1 - T_0(\lfloor \frac{h 2^{n+1}}{N} \rfloor) \right] \right] \bmod N \tag{3}$$

Relying on (3), our algorithm would require an additional nested loop in order to compute the sum, iterating over bits of h for each output pair. However, because our algorithm iterates over the output array in ascending order, evaluation of $I_N(h)$

DRAFT

using a recurrence relation is a possible approach. Reusing computations from the previous iteration actually allows to perform fewer operations than direct evaluation of the above sum. Consider the sequence:

$$K_N(h) = L - C(h) - 1 \tag{4}$$

This yields $K_{32} = (4, 3, 4, 2, 4, 3, 4, 1, 4, 3, 4, 2, 4, 3, 4, 0, 4, 3, 4, 2, 4, 3, 4, 1, 4, 3, 4, 2, 4, 3, 4, ...)$. We observe in Fig.5 that terms $2^{K_{32}(h)}$ appear in $I_{32}(h+1)$ when $2^{K_{32}(h)-1}$ is not in $I_{32}(h)$. Moreover, existing $2^{K_{32}(h)-1}$ terms in $I_{32}(h)$ are negated in $I_{32}(h+1)$. Finally, all terms with a magnitude greater than $2^{K_{32}(h)}$ are cleared in $I_{32}(h+1)$. Based on these observations, we propose a recurrence relation that is valid for all N. For this purpose, we separate the positive and negative terms. Additionally, we avoid cases where $2^{K_N(h)-1} = 2^{-1}$ by doubling all terms. We also need to extract binary digits of positive terms:

$$\begin{aligned} I_N(h) &= \frac{P_N(h) - Q_N(h)}{2} \mod N \\ P_N(h) &= \sum_{n=0}^{L-1} P_N(h)_n 2^n \end{aligned} \tag{5}$$

Based on (4), (5) and our observations, the proposed recurrence relation can be defined as:

$$\begin{cases} P_N(0) & = 0 \\ Q_N(0) & = 0 \\ P_N(h) & = P_N(h-1) \mod 2^{K_N(h)} \\ & \quad + \left[1 - P_N(h-1)_{K_N(h)}\right] 2^{K_N(h)+1} \\ Q_N(h) & = Q_N(h-1) \mod 2^{K_N(h)} \\ & \quad + P_N(h-1)_{K_N(h)} 2^{K_N(h)} \end{cases} \tag{6}$$

The depth-first iterative CPFFT algorithm presented in Fig.6 relies on the butterfly scheduling mechanism proposed in the previous section and employs the recurrence relation (6) for computing input indices. An implementation of the algorithm with additional twiddle factor sharing is proposed in the Appendix.

Because our algorithm iterates over output pairs, processing base cases of length 2, the recurrence relation (6) is actually used to compute $I_N(g)$ rather than $I_N(h)$. This is convenient since it allows reusing the value of $C(g)$, involved in butterfly scheduling. In the same way, it is also possible to use this recurrence to generate indices for larger base cases, allowing usual FFT optimizations. When a sub-transform that processes a base case of length m is to be used in place of the radix-2 butterfly, a pair of sub-transforms of lengths $m/2$ have to be computed in the alternative branch of the conditional.

Since only powers of 2 are involved, implementation of the recurrence can be done by inversion and copy of bits between two variables p and q, where bit positions are derived from $C(g)$. Again, this is cheap to perform in hardware and requires a few bit-oriented instructions in software.

V. Experimental evaluation

Because the proposed algorithm structure does not impact the arithmetic complexity of CPFFT, a new evaluation of the arithmetic operations count is not required. However, the novel structure impacts the memory access behavior of the algorithm. When conceiving an FFT algorithm, there are multiple design choices that impact both the number of memory accesses performed and the access pattern. In this section we will explore a subset of the FFT algorithm design space in order to determine how the proposed algorithm compares to others when accessing memory. The first part of the experiment focuses on memory bandwidth and the second part on cache friendliness.

function CPFFT_DI$^S_N(out[N], in[N])$
  $p, q \leftarrow 0, 0$
  for $g \leftarrow 0$ **to** $N/2 - 1$ **do**
    $c \leftarrow \lfloor \log_2(g \oplus (g+1)) \rfloor$
    $i_0 \leftarrow (p - q)/4 \bmod N$
    $i_1 \leftarrow (i_0 + N/2) \bmod N$
    if $c \bmod 2 = 1$ **then**
      $out[2g], out[2g+1] \leftarrow in[i_0], in[i_1]$
    else
      CP_BF2$(out, 2g, in, i_0, i_1)$
    for $j \leftarrow 1 - c \bmod 2$ **to** $c - 1$ **by** 2 **do**
      for $b \leftarrow 0$ **to** $2^j - 1$ **do**
        CP_BF4$^S(2^{j+2}, out, 2g - 2^{j+2} + 2, b)$
    $k \leftarrow \log_2(N) - c - 1$
    $q \leftarrow q \bmod 2^k + (\text{bit } k \text{ of } p)2^k$
    $p \leftarrow p \bmod 2^k + (1-\text{bit } k \text{ of } p)2^{k+1}$

Fig. 6. The proposed depth-first iterative CPFFT algorithm

The explored design space contains implementations of out-of-place DIT FFT algorithms. The four base algorithms considered are: the CTFFT radix-2 algorithm, the mixed radix-2/4 FFT algorithm, the SRFFT algorithm and the CPFFT algorithm. The resulting implementations are denoted CT, MR, SR and CP respectively. These algorithms are implemented using three different structures that impact the datapath traversal order: the breadth-first iterative structure, the depth-first recursive structure and the proposed depth-first iterative structure, denoted BI, DR and DI respectively.

In addition to the memory accesses generated by the compiler for storage of the program's local variables, FFT implementations perform explicit accesses to memory for distinct purposes: accessing samples in the input and output arrays, reading from the twiddle factor table and optionally performing recursive calls and loads from additional lookup tables. Along with the datapath traversal order, different array access strategies exist that greatly impact the memory behavior of FFT algorithm implementations.

Among the multiple twiddle access optimization strategies described in the literature, we retained three that can be readily applied to the four base algorithms considered in our experiment. First, depending on the design of the butterflies, the memory bandwidth may be reduced by loading a single twiddle that can be used to compute related twiddles by trivial operations [11], [23]. This allows loading a single twiddle table entry shared by multiple butterflies in the block. This twiddle factor sharing (S) strategy is applicable to all evaluated

	DI DR BI	DI_F DR_F BI_F	DI_S DR_S BI_S	DI_SF DR_SF BI_SF	BI_G	BI_GF	BI_GS	BI_GSF
CT i/o refs	$2.000 \cdot NL$	$2.000 \cdot NL$	$2.000 \cdot NL$		$2.000 \cdot NL$	$2.000 \cdot NL$	$2.000 \cdot NL$	
CT tw loads	$0.500 \cdot NL'$	$0.500 \cdot NL'$	$0.125 \cdot NL'$		$1.000 \cdot N$	$1.000 \cdot N$	$0.250 \cdot N$	
CT tw entries	$0.500 \cdot N$	$0.125 \cdot N$	$0.125 \cdot N$		$0.500 \cdot N$	$0.125 \cdot N$	$0.125 \cdot N$	
MR i/o refs	$1.000 \cdot NL$	$1.000 \cdot NL$	$1.000 \cdot NL$	$1.000 \cdot NL$	$1.000 \cdot NL$	$1.000 \cdot NL$	$1.000 \cdot NL$	$1.000 \cdot NL$
MR tw loads	$0.375 \cdot NL'$	$0.375 \cdot NL'$	$0.187 \cdot NL'$	$0.187 \cdot NL'$	$1.000 \cdot N$	$1.000 \cdot N$	$0.500 \cdot N$	$0.500 \cdot N$
MR tw entries	$1.000 \cdot N$	$0.125 \cdot N$	$0.500 \cdot N$	$0.125 \cdot N$	$1.000 \cdot N$	$0.125 \cdot N$	$0.500 \cdot N$	$0.125 \cdot N$
SR i/o refs	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$
SR tw loads	$0.333 \cdot NL'$	$0.333 \cdot NL'$	$0.167 \cdot NL'$	$0.167 \cdot NL'$	$1.000 \cdot N$	$1.000 \cdot N$	$0.500 \cdot N$	$0.500 \cdot N$
SR tw entries	$0.500 \cdot N$	$0.125 \cdot N$	$0.250 \cdot N$	$0.125 \cdot N$	$0.500 \cdot N$	$0.125 \cdot N$	$0.250 \cdot N$	$0.125 \cdot N$
SR_L LUT refs	$1.351 \cdot N$	$1.351 \cdot N$	$1.351 \cdot N$	$1.351 \cdot N$	$1.278 \cdot N+$ $0.083 \cdot NL$	$1.278 \cdot N+$ $0.083 \cdot NL$	$1.278 \cdot N+$ $0.083 \cdot NL$	$1.278 \cdot N+$ $0.083 \cdot NL$
CP i/o refs	$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$		$1.333 \cdot NL$	$1.333 \cdot NL$	$1.333 \cdot NL$	
CP tw loads	$0.167 \cdot NL'$	$0.167 \cdot NL'$	$0.083 \cdot NL'$		$0.500 \cdot N$	$0.500 \cdot N$	$0.250 \cdot N$	
CP tw entries	$0.250 \cdot N$	$0.125 \cdot N$	$0.125 \cdot N$		$0.250 \cdot N$	$0.125 \cdot N$	$0.125 \cdot N$	
CP_L LUT refs	$1.351 \cdot N$	$1.351 \cdot N$	$1.351 \cdot N$		$0.823 \cdot N+$ $0.166 \cdot NL$	$0.823 \cdot N+$ $0.166 \cdot NL$	$1.046 \cdot N+$ $0.082 \cdot NL$	

Fig. 7. Estimates for the number of memory references to the input/output arrays, the number of memory loads from the twiddle table, the number of twiddle entries and the number of references to additional lookup tables. These estimates can all be expressed as products of the transform length N and related parameters $L = log_2(N)$ and $L' = log_2(N/8)$. The values were measured for all evaluated algorithms and grouped by equivalent results. The LUT references results are only relevant for implementations that rely on lookup tables for butterfly scheduling and input indexing.

algorithms to different extents, allowing to divide the loads count and the number of stored twiddle entries by 2, 4 or 8.

Then, symmetry of the sine function always allows to keep only $N/8$ roots of unity in the twiddle table. When twiddle sharing is not implemented or when it is unable to reduce the table length down to $N/8$, exploiting the sine symmetry can still be done by folding (F) the table index on each twiddle access. This however does not reduce the number of loads performed by the algorithm and requires conditional manipulations of both the loaded twiddle value and the twiddle index [24], [25].

Finally, when the breadth-first approach is implemented, it is possible to exchange the inner loops in order to group (G) all the butterflies that use the same twiddles in a stage [26], further reducing the number of related loads. It is worth noting that both S and G twiddle access strategies impose constraints on the algorithm structure that also impact the access order to the array of samples.

In addition to the memory accesses mentioned above, some implementations need to perform even more loads as algorithms are designed to rely on two or three lookup tables (L) in order to implement the complex datapath structure of SRFFT and CPFFT. When such a design choice is retained for these algorithms, lookup tables are used to store the butterfly scheduling pattern and optionally the permutation of input sample indices. The scheduling pattern is described by two tables that contain the number of butterflies to compute for each stage as well as the indices that require computation, as proposed in [6], [13].

We retained 47 FFT algorithm implementations formed by combination of the mentioned base algorithms (denoted CT, MR, SR and CP) and the various design choices mentioned previously (denoted BI, DR, DI, S, G, F and L). Most of the resulting algorithm structures are of common use and some have been described by other authors. We have implemented the following variants: CP_BI_F_L, CP_BI_GF_L, CP_BI_GS_L, CP_BI_G_L, CP_BI_L [13], CP_BI_S_L, CP_DI, CP_DI_F, CP_DI_S, CP_DR [12], CP_DR_F, CP_DR_S, CT_BI, CT_BI_F, CT_BI_G, CT_BI_GF, CT_BI_GS, CT_BI_S, CT_DR, CT_DR_F, CT_DR_S, MR_BI, MR_BI_F, MR_BI_G [26], MR_BI_GF, MR_BI_GS, MR_BI_GSF, MR_BI_S, MR_BI_SF, MR_DR, MR_DR_F, MR_DR_S, MR_DR_SF, SR_BI_F_L, SR_BI_G [7], SR_BI_GF, SR_BI_GF_L, SR_BI_GSF_L, SR_BI_GS_L, SR_BI_G_L [6], SR_BI_L, SR_BI_SF_L, SR_BI_S_L, SR_DR [12], SR_DR_F, SR_DR_S and SR_DR_SF.

All evaluated implementations have been written in the C language with uniform programming practices. They all use hard-coded base cases of length 4 which process double precision samples. Accessing the twiddle table for $\omega = e^0$ and $\omega = e^{-\mathrm{i}\pi/4}$ is avoided by relying on simplified butterfly functions when appropriate. The source code of the implementations is made available under a free software license [27].

The number of sample and twiddle-related memory accesses performed by the implementations under evaluation were measured by running instrumented programs, that is counters introduced in the source code. The programs were executed for transform lengths between 2^4 and 2^{24}. The obtained results are proportional to N, $\log N$, $\log N/8$ or a mix thereof. We thus were able to use a fitting method to extract the constant factors shown in Fig.7. This table gives estimates for memory access counts expressed as functions of the transform length for all evaluated algorithms.

We observe that the number of sample-related accesses varies with the base algorithm only. The MR implementations yield the lowest value. Because samples are stored and loaded

DRAFT

multiple times as the output array is used to keep intermediate results, this measurement is expected to change with the size of the hard-coded base case. Likewise, it does not account for the compiler ability to keep some of the intermediate results in registers.

On the contrary, the number of twiddle-related loads differs between variants of the same base algorithm. The CP_S implementations have the lowest number of twiddle-related accesses of any depth-first algorithm. This is due to the CPFFT algorithm requiring only a single twiddle value per butterfly and a $N/4$ sized twiddle table. Twiddle sharing is then sufficient to halve the number of loads and reduce the table size down to the minimal length of $N/8$. This makes the more costly folding strategy (F) irrelevant with CPFFT. In contrast, breadth-first approaches allow implementation of grouping (G) which further divides the number of twiddle-related loads by $\log N/8$, allowing CT_BI_GS and CP_BI_GS implementations to yield the lowest number of twiddle loads when N is large enough.

We also observe that CP_L and SR_L implementations perform a number of additional memory loads from lookup tables proportional to N. When combined with grouping (G), an additional term proportional to $NlogN$ is introduced. Indeed, exchanging the inner loops in order to reduce the number of twiddle loads has the opposite effect on the number of scheduling-related loads.

As mentioned previously, when memory caches are involved, the number of memory accesses performed by the implementation is not directly related to the main memory bandwidth. Depending on the platform architecture, the number of memory cache misses generated by the algorithm can have a major performance impact. Many factors have to be considered when optimizing for a cache-based architecture. Regarding the evaluated implementations, the obtained results suggest that the size of the precomputed tables, the call stack usage and the array access patterns all contribute to the amount of cache misses.

High performance FFT software packages rely on hard-coded base cases larger than what is implemented in our experiment. However, we believe that increasing the length of the base case does not have a major impact on the data cache measurements because this optimization aims to remove highly localized accesses to intermediate results. When recursive code is optimized by compilers, the generated DR programs actually implement base cases that are larger than expressed at the source level, due to inlining of leaf function calls. We have observed that disabling this optimization does not yield results significantly different from what is presented below.

In order to measure the cache behavior of the 47 implementations, we relied on a virtual machine that embeds a cache simulation model. The *valgrind* project provides such a tool [28]. It is designed to execute native binary code while performing various run-time analysis. The *callgrind* module is able simulate two levels of cache while recording the number of hits and misses [29]. The chosen simulated cache sizes were 16384 bytes for the first level and 1048576 bytes for the second level. Both caches have been parameterized to use a line size of 32 and an associativity of 8. The evaluated programs are 64-bit *x86* binaries generated by using the `-O3 -ffast-math` compiler options. Unlike with the previous experiment relying on source level embedded counters, the quality of the generated code impacts the results. In order to mitigate the impact of possible compiler miss-optimizations on the measurements, the programs have been compiled with three different compilers and the best performing program file has been retained for each FFT implementation. The compilers used were the *GNU Compiler Collection* version 9.2.0, the *LLVM*-based *Clang* compiler version 9.0.0 and the *Intel C++ Compiler* version 19.0.5.281. Measurements were only performed during execution of the FFT algorithm in the program, excluding all initialization and reporting steps that take place before and after the actual processing. The simulated cache is cold on FFT algorithm entry.

The measured number of cache misses per sample is plotted for all algorithms in Fig.8. The plot shapes are similar for the two levels of cache. We can see that as long as the length of the transform is small enough, all algorithms have similar cache performance. As expected, when the cache becomes too small, the number of misses per sample starts to rise. This occurs either progressively or by suddenly jumping to high values, depending on the algorithm. Then, as the transform length is further increased, all implementations generate more misses but with different progression rates. Analysis of the data also confirms that for large transform lengths, depth-first implementations perform better than breadth-first implementations. In order to display results of the best performing implementations in more detail, measurements were sorted by number of cache misses separately for different transform lengths. The five best performing implementations are listed in Fig.9 for five different transform lengths between 2^8 and 2^{24}. We notice that implementations relying on additional lookup tables (L) do not appear in any of the truncated lists. No such implementation is actually ranked better than 14 in any list. We also notice that no recursive implementations appear for small transform lengths and no breadth-first implementations appear for large transform lengths. This does not occur before rank 10 with MR_DR_F and rank 11 with MR_BI_SF for lengths 2^8 and 2^{24} respectively. In contrast, the CP_DI_S implementation, based on the proposed depth-first iterative approach, is ranked between 1 and 4 in all lists. Moreover, CP_DI_S has the lowest amount of cache misses for large transforms on the two caches.

Nowadays, high performance processors implement another specific cache component designed to optimize execution of conditional branches. Because the complexity of the retained FFT datapath shape drives the algorithm control flow, it is expected that irregular datapath structures impact the performance on processors that implement a branch predictor. Even if there are multiple branch predictor design strategies that yield different behaviors and performance, the predictor model provided by the *valgrind* tool can give an insight of the possible performance penalty incurred by a complex datapath structure. The measured number of conditional branch mispredictions per sample is plotted for all algorithms in Fig.10. We can see that when N is large enough, the results do not depend on the transform length. All branch predictor-related results discussed here are related to transforms of length 2^{24}.

DRAFT

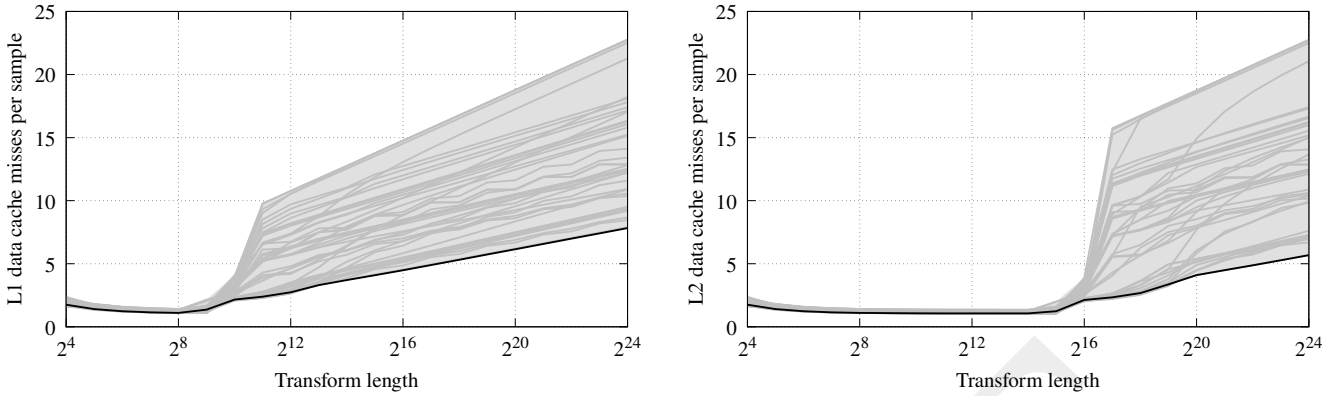

Fig. 8. Measured number of data cache misses per sample for all evaluated algorithms. The left and right plots are related to the level 1 and level 2 caches respectively. Results related to the CP_DI_S algorithm are emphasized using black line plots.

	$N = 2^8$	$N = 2^{12}$	$N = 2^{16}$	$N = 2^{20}$	$N = 2^{24}$
#1	MR_BI_F	MR_DR_F	**CP_DI_S**	**CP_DI_S**	**CP_DI_S**
	1.09766	2.63037	4.48425	6.1502	7.82717
#2	CT_BI_F	MR_DR_SF	CP_DR_S	CP_DR_S	CP_DR_S
	1.10156	2.65625	4.535	6.20647	7.89299
#3	MR_BI_GF	**CP_DI_S**	MR_DR_SF	MR_DR_SF	MR_DR_SF
	1.10156	2.73779	4.86038	6.67025	8.45642
#4	**CP_DI_S**	CP_DR_S	MR_DR_S	MR_DR_S	MR_DR_S
	1.10547	2.80908	5.11118	6.8849	8.66334
#5	CT_BI_S	CP_DI_F	CP_DI_F	CP_DI_F	CP_DI_F
	1.10547	2.84253	5.17249	7.19755	9.19956

	$N = 2^8$	$N = 2^{12}$	$N = 2^{16}$	$N = 2^{20}$	$N = 2^{24}$
#1	MR_BI_F	MR_BI_F	MR_DR_SF	**CP_DI_S**	**CP_DI_S**
	1.09766	1.0647	2.02948	4.09885	5.68393
#2	CT_BI_F	CT_BI_F	MR_DR_F	CP_DR_S	CP_DR_S
	1.10156	1.06494	2.06219	4.10009	5.68511
#3	MR_BI_GF	MR_BI_GF	**CP_DI_S**	MR_DR_SF	MR_DR_SF
	1.10156	1.06494	2.12862	4.47248	6.68751
#4	**CP_DI_S**	**CP_DI_S**	CP_DR_S	CP_DR_F	MR_DR_S
	1.10547	1.06519	2.13014	4.47486	6.95293
#5	CT_BI_S	CT_BI_S	CT_DR_S	CP_DI_F	CP_DR_F
	1.10547	1.06519	2.13135	4.48298	6.98917

Fig. 9. Algorithms sorted according to the number of data cache misses measured while computing transforms of various lengths. The left and right tables are related to the level 1 and level 2 caches respectively. Only five algorithms that yield the lowest number of misses per sample are displayed for each transform length.

The MR_BI_S implementation yields a value as low as 0.005 misses per sample. It is worth noting that when excluding breadth-first algorithms that do not perform well with the data cache on large transforms, the MR_DR_S implementation has the best branch predictor behavior with a miss rate of 0.032. There are only MR and CT implementations ranked between 1 and 18. The fact that no SR and CP implementations are ranked better than 19 tends to acknowledge the initial hypothesis. All SR and CP programs have miss rates between 0.106 and 0.408 with the proposed CP_DI_S algorithm yielding 0.109.

Different processor architectures have different penalties in terms of number of cycles for data cache misses and branch mispredictions. Instead of performing global execution time measurements related to a specific architecture, we choose to provide separate cache-related results. Other experiments and measurements could be considered in order to better understand the impact of the algorithm structures on performance. However, we believe the provided cache-related results help to forecast behavior of the algorithms on various architectures.

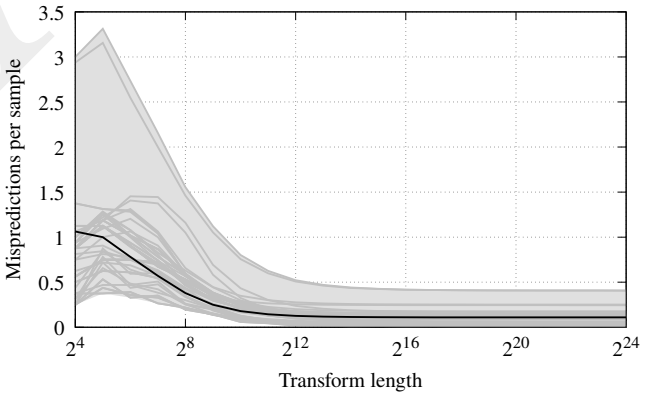

Fig. 10. Measured number of branch mispredictions per sample for all evaluated algorithms. Results related to the CP_DI_S algorithm are emphasized using a black line plot.

VI. Conclusion

Depth-first FFT algorithms are favored in state-of-the-art implementations because of the cache-friendly order they use to access memory. Even if the depth-first order does not imply recursion, the two concepts are not always clearly distinguished in the literature. A recursive algorithm relies on memory to implement the depth-first order by constantly accessing a stack data structure. In this paper, we proposed a novel approach to perform the depth-first traversal of the SRFFT datapath iteratively with $\mathcal{O}(1)$ space complexity, based on properties of binary numbers. Moreover, our approach allows on the fly generation of input indices for CPFFT. Although the proposed iterative algorithm is not straightforward, it is still concise as it uses a few arithmetic and bitwise operations in place of multiple recursions.

It has previously been noticed that the *binary carry sequence* is related to the optimal solution of the well-known

DRAFT

Tower of Hanoi mathematical game [30]. There is a concise recursive algorithm that yields this solution. In this paper, we have shown that SRFFT is another recursive algorithm that can be related to the *binary carry sequence*. This strongly suggests that the sequence is actually not specific to either of these two algorithms but is instead related to their recursive structure. Relying on this sequence to devise iterative variants of recursive divide and conquer algorithms, possibly unrelated to the FFT, is thus a research topic of interest.

Future work may also include design, optimization and evaluation of both hardware and software implementations of the proposed FFT algorithm.

VII. Acknowledgment

The authors would like to thank Alexandra Zaharia for comments that improved the manuscript and Mehdi Khairy for the valuable discussions we had.

Appendix
C program for the depth-first iterative CPFFT

```c
#include <complex.h>
#include <math.h>
#include <stdint.h>
#include <stdlib.h>

/* Initialization of the twiddles table */

complex double *
cpfft_init(unsigned n)
{
  unsigned m = n / 8;
  complex double *tw = malloc(m * sizeof(*tw));

  if (tw) {
    for (unsigned i = 0; i < m; i++) {
      tw[i] = cexp(-2 * M_PI * I * i / n);
    }
  }

  return tw;
}

/* Conjugate Pair FFT radix-4 butterfly */

static inline void
cp_bf4(unsigned s,
       complex double * restrict out,
       complex double w)
{
  complex double a = out[0];
  complex double b = out[s];
  complex double c = out[s * 2];
  complex double d = out[s * 3];
  out[0]     = a +     (w * c + conj(w) * d);
  out[s]     = b - I * (w * c - conj(w) * d);
  out[s * 2] = a -     (w * c + conj(w) * d);
  out[s * 3] = b + I * (w * c - conj(w) * d);
}

/* Depth-First Iterative Conjugate Pair FFT */

void
cpfft_di(unsigned n,
         complex double * restrict out,
         const complex double * restrict in,
         const complex double * restrict tw)
{
  unsigned log2_n = 31 - __builtin_clz(n);
  unsigned r = 32 - log2_n;
  uint32_t p = 0, q = 0;    /* allows n <= 1<<31 */

  /* output indices */
  for (uint32_t h2, h = 0; h < n; h = h2) {

    /* binary carry sequence */
    h2 = h + 2;
    unsigned c = 30 - __builtin_clz(h ^ h2);

    /* input indices */
    unsigned i0 = (p - q) >> r;
    unsigned i1 = i0 ^ (n >> 1);

    if (c & 1) {                /* stage 1 */
      out[h    ] = in[i0];
      out[h + 1] = in[i1];
      cp_bf4(1, out + h - 2, 1);

    } else {                    /* stage 0 */
      out[h    ] = in[i0] + in[i1];
      out[h + 1] = in[i0] - in[i1];
    }

    /* higher stages */
    for (unsigned j = 1 + (c & 1); j < c; j += 2) {
      unsigned s = 1 << j;
      unsigned r = h2 - 4 * s;
      unsigned t = log2_n - j - 2;

      /* butterfly blocks */
      for (unsigned b = 1; b < s / 2; b++) {
        /* w = cexp(-2 * M_PI * I * b / s / 4); */
        complex double w = tw[b << t];

        cp_bf4(s, out + r + b, w);
        cp_bf4(s, out + r + s - b, conj(w) * -I);
      }

      cp_bf4(s, out + r, 1);
      cp_bf4(s, out + r + s/2, M_SQRT1_2 * (1-I));
    }

    /* next input index */
    uint32_t m2 = 0x20000000 >> c;
    uint32_t m1 = m2 - 1;
    uint32_t m  = p & m2;
    q = (q & m1) | m;
    p = (p & m1) | ((m ^ m2) << 1);
  }
}
```

References

[1] S. G. Johnson and M. Frigo, "Implementing FFTs in Practice ∗," 2009.
[2] R. Yavne, "An economical method for calculating the discrete Fourier transform," *ACM*, pp. 115–125, Dec 1968.
[3] P. Duhamel and H. Hollmann, "Split radix FFT algorithm," *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, Jan 1984.
[4] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
[5] P. Duhamel and M. Vetterli, "Fast fourier transforms: A tutorial review and a state of the art," *Signal Processing (Elsevier)*, vol. 19, no. 4, pp. 259–299, 1990.

DRAFT

[6] A. N. Skodras and A. G. Constantinides, "Efficient computation of the split-radix FFT," *IEE Proceedings F - Radar and Signal Processing*, vol. 139, no. 1, pp. 56–60, Feb 1992.

[7] H. Sorensen, M. Heideman, and C. Burrus, "On computing the split-radix FFT," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 34, no. 1, pp. 152–156, Feb 1986.

[8] I. Kamar and Y. Elcherif, "Conjugate pair fast Fourier transform," *Electron. Lett.*, vol. 25, no. 5, pp. 324–325, Mar 1989.

[9] A. M. Krot and H. B. Minervina, "Comment on 'Conjugate pair fast Fourier transform'," *Electron. Lett.*, vol. 28, no. 12, pp. 1143–1144, Jun 1992.

[10] H.-S. Quian and H.-J. Zhao, "Comment on Conjugate pair fast Fourier transform," *Electron. Lett.*, vol. 26, no. 8, pp. 541–542, Apr 1990.

[11] S. G. Johnson and M. Frigo, "A Modified Split-Radix FFT With Fewer Arithmetic Operations," *IEEE Trans. Signal Process.*, vol. 55, no. 1, pp. 111–119, Jan 2007.

[12] A. M. Blake, "Computing the fast fourier transform on simd microprocessors," Ph.D. dissertation, University of Waikato, Hamilton, New Zealand, 2012, thesis, Doctor of Philosophy (PhD).

[13] S. Ocovaj and Z. Lukac, "Optimization of conjugate-pair split-radix FFT algorithm for SIMD platforms," *IEEE International Conference on Consumer Electronics (ICCE)*, pp. 373–374, Jan 2014.

[14] R. C. Singleton, "On computing the fast Fourier transform," *Commun. ACM*, vol. 10, no. 10, pp. 647–654, Oct 1967.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-Oblivious Algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 1–22, Jan 2012.

[16] A. M. Blake, I. H. Witten, and M. J. Cree, "The Fastest Fourier Transform in the South," *IEEE Trans. Signal Process.*, vol. 61, no. 19, pp. 4707–4716, Oct 2013.

[17] S.-J. Lin and W.-H. Chung, "The split-radix fast Fourier transforms with radix-4 butterfly units," *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, Oct 2013.

[18] A. Blake and M. Hunter, "Dynamically Generating FFT Code," *J. Signal Process. Syst.*, vol. 76, no. 3, pp. 275–281, Sep 2014.

[19] W. Zheng, K. Li, and K. Li, "Scaled Radix-2/8 Algorithm for Efficient Computation of Length-$N = 2^m$ DFTs," *IEEE Trans. Signal Process.*, vol. 62, no. 10, pp. 2492–2503, Mar 2014.

[20] N. J. A. S. Karamanos Konstantinos, "A035263, The On-Line Encyclopedia of Integer Sequences," 2000, parity of 2-adic valuation of n. [Online]. Available: http://oeis.org/A035263

[21] J. Tromp, "A007814, The On-Line Encyclopedia of Integer Sequences," 1996, exponent of highest power of 2 dividing n, a.k.a. the binary carry sequence, the ruler sequence, or the 2-adic valuation of n. [Online]. Available: http://oeis.org/A007814

[22] D. Evans, "An improved digit-reversal permutation algorithm for the fast Fourier and Hartley transforms," *IEEE Trans. Acoust. Speech Signal Process.*, vol. 35, no. 8, pp. 1120–1125, Aug 1987.

[23] Y. Wang, Y. Tang, Y. Jiang, J.-G. Chung, S.-S. Song, and M.-S. Lim, "Novel Memory Reference Reduction Methods for FFT Implementations on DSP Processors," *IEEE Trans. Signal Process.*, vol. 55, no. 5, pp. 2338–2349, Apr 2007.

[24] F. Qureshi and O. Gustafsson, "Analysis of twiddle factor memory complexity of radix-2i pipelined FFTs," *Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, pp. 217–220, Nov 2009.

[25] M. Hasan and T. Arslan, "Scheme for reducing size of coefficient memory in FFT processor," *Electron. Lett.*, vol. 38, no. 4, pp. 163–164, Feb 2002.

[26] G. H. Allen, "Programming an efficient radix-four FFT algorithm," *Signal Process.*, vol. 6, no. 4, pp. 325–329, Aug 1984.

[27] A. Becoulet, "A Collection of FFT Algorithms." [Online]. Available: https://github.com/diaxen/fft-garden

[28] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, Jun 2007.

[29] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A Tool Suite for Simulation Based Analysis of Memory Access Behavior," *SpringerLink*, pp. 440–447, Jun 2004.

[30] E. W. Weisstein, "Binary Carry Sequence." [Online]. Available: http://mathworld.wolfram.com/BinaryCarrySequence.html

DRAFT