EURASIP Journal on Advances in Signal Processing

RESEARCH Open Access

Instruction scheduling heuristic for an efficient FFT in VLIW processors with balanced resource usage

Mounir Bahtat[1*], Said Belkouch[1], Philippe Elleaume[2] and Philippe Le Gall[2]

Abstract

The fast Fourier transform (FFT) is perhaps today's most ubiquitous algorithm used with digital data; hence, it is still being studied extensively. Besides the benefit of reducing the arithmetic count in the FFT algorithm, memory references and scheme's projection on processor's architecture are critical for a fast and efficient implementation. One of the main bottlenecks is in the long latency memory accesses to butterflies' legs and in the redundant references to twiddle factors. In this paper, we describe a new FFT implementation on high-end very long instruction word (VLIW) digital signal processors (DSP), which presents improved performance in terms of clock cycles due to the resulting low-level resource balance and to the reduced memory accesses of twiddle factors. The method introduces a tradeoff parameter between accuracy and speed. Additionally, we suggest a cache-efficient implementation methodology for the FFT, dependently on the provided VLIW hardware resources and cache structure. Experimental results on a TI VLIW DSP show that our method reduces the number of clock cycles by an average of 51 % (2 times acceleration) when compared to the most assembly-optimized and vendor-tuned FFT libraries. The FFT was generated using an instruction-level scheduling heuristic. It is a modulo-based register-sensitive scheduling algorithm, which is able to compute an aggressively efficient sequence of VLIW instructions for the FFT, maximizing the parallelism rate and minimizing clock cycles and register usage.

Keywords: FFT, VLIW, Scheduling heuristic, Twiddle factors, Modulo scheduling, Software pipelining

1 Introduction

The discrete Fourier transform (DFT) is a used transform for spectral analysis of finite-domain discrete-time signals. It is widely employed in signal processing systems. For decades, studies have been done to improve the algorithmic efficiency of this technique and this is still an active research field. The frequency response of a discrete signal $x[n]$ can be computed using the DFT formula over N samples as in (1).

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} \qquad (1)$$

For $k \in \{0,\ldots,N-1\}$ where $W_N^{nk} = e^{-j(2\pi/N)nk}$

The DFT algorithmic complexity is $O(N^2)$. In order to reduce this arithmetic count and therefore enhancing its implementation efficiency, a set of methods were proposed. These methods are commonly known as fast Fourier transforms (FFTs), and they present a valuable enhancement in complexity of $O(N\log(N))$. FFT was first discovered by Gauss in the eighteenth century and re-proposed by Cooley and Tukey in 1965 [1]. The idea is based on the fundamental principle of dividing the computation of a DFT into smaller successive DFTs and recursively repeating this process. The fixed-radix category of FFT algorithms mainly includes radix-2 (dividing the DFT into 2 parts), radix-4 (into 4 parts), radix-2^2, and radix-8 [2]. Mixed-radix FFTs combine several fixed-radix algorithms for better convenience [3]. Split-radix FFTs offer lower arithmetic count than the fixed or mixed-radix, using a special irregular decomposition [4, 5]. Also, a recursive FFT can be implemented as in [6], and a

* Correspondence: mnr.bahtat@gmail.com
[1]LGECOS Lab, ENSA-Marrakech of the Cadi Ayyad University, Marrakech, Morocco
Full list of author information is available at the end of the article

combination between the decimation-in-frequency (DIF) and the decimation-in-time (DIT) FFT is proposed in [7].

The FFT algorithm has been implemented for either hardware or software, on different platforms. Hardware IP implementations on ASIC or FPGA can provide real-time high-speed solutions but lack flexibility [8]. Equivalent implementations on general purpose processors (GPP) offer flexibility, but it is generally slower and cannot meet high real-time constraints. Multi-core digital signal processors (DSPs) are interesting hardware platforms achieving the tradeoff between flexibility and performance. These are sharing with FPGAs a well-earned reputation of being difficult for developing parallel applications. As a result, several languages (such as OpenCL) seeking to exploit the power of multi-cores while remaining platform independent has been recently explored [9, 10]. OpenCL for instance is an industry's attempt to unify embedded multi-core programming, allowing data parallelism, SIMD instructions, and data locality as well [11].

Modern low-power multi-core DSP architectures attracted many real-time applications with power restrictions. One of the primary examples in this field is the C6678 multi-core DSP from Texas Instruments, which can provide up to 16 GFLOPS/watt. In [12], a real-time low-power motion estimation algorithm based of the McGM gradient model has been implemented in the TI C6678 DSP, exploiting DSP features and loop-level very long instruction word (VLIW) parallelism. The implementation provided significant power efficiency gains toward high-end current architectures (multi-core CPUs, many-core GPUs). In [13], a low-level optimization of the 4-, 8-, and 16-point FFTs in the C6678 DSP is presented.

The most recent high-end DSP architectures are VLIW, which mainly support an instruction-level parallelism (ILP) feature, offering the possibility to execute simultaneously multiple instructions and a data-level parallelism allowing the access to multiple data during each cycle. Therefore, these kinds of processors are known to have greater performance compared to RISC or CISC, even having simpler and more explicit internal design. However, unlike superscalar processors where parallelism opportunities are discovered by hardware at the run time, VLIW DSPs leave this task to the software compiler. It has been shown that constructing an optimal compiler performing instruction mapping and scheduling in VLIW architectures is an NP-Complete problem [14]. Toward this increasingly difficult task, compilers have been unable to capitalize on these existing features and often cannot produce efficient code [15].

Several compilation techniques tackled this problem. A classical method is called the list scheduling [16] and it schedules instructions within a single block, by building a directed acyclic dependency graph (DAG) (as in Fig. 1). Trace scheduling was introduced afterwards by Fisher [17], which selects and acts on acyclic paths having the highest probability of execution (called traces). The trace scheduling was unable to properly optimize loops; then, a more effective solution to this specific problem was proposed by software pipelining [18]. The most successful technique of software pipelining is known as the modulo scheduling [19]. We have proposed an efficient enumeration-based heuristic for modulo scheduling in [20].

One of the bottlenecks toward an efficient FFT implementation on these VLIW DSP platforms is memory latencies. Indeed, in addition to the memory access of

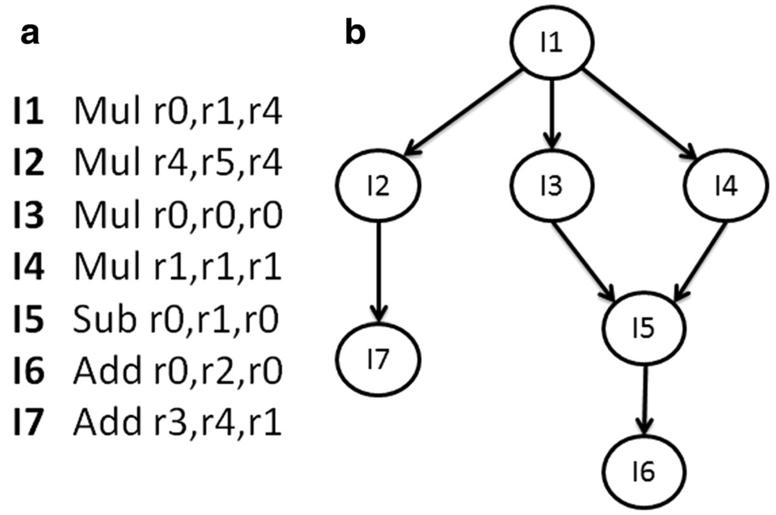


Fig. 1 Dependency graph (**b**) of an instruction block (**a**)

butterflies' inputs and outputs over several stages ($\log_2(N)$ stages for a radix-2 FFT, shown in Fig. 2), conventional algorithms excessively and redundantly load twiddle factors (W_N^k). These latter are classically stored as a N-sized complex vector, requiring $O(N\log(N))$ accesses. In [21], a novel memory reference reduction method is introduced by grouping butterflies using the same twiddle factors together; therefore decreasing the number of memory references due to twiddle factors in DSPs by 76 %. Another FFT scheme is presented in [22], reducing the memory access frequency and multiplication operations. Also, other results in [23] decrease the number of twiddle accesses by an asymptotic factor of $\log(N)$, based on G and G^T transforms. On the other hand, the FFT performance is tightly dependent to the processor's architecture and its memory and cache structure. Kelefouras et al. [24] propose a methodology to speed up the FFT algorithm depending on the memory hierarchy of processor's architecture. State-of-the-art FFT libraries such as FFTW [25–27] and UHFFT [28] maximize the performance by adapting to the hardware at run time, usually using a planner, searching over a large space of parameters in order to pick the best implementation. The FFTS in [29] claims an efficient cache-oblivious FFT scheme that is not requiring any machine-specific calibration.

In the present paper, we propose an efficient modulo-like FFT scheduling for VLIW processors. Our implemented methodology allows better core-level resource balance, exploiting the fact that the twiddle factors can be calculated recurrently using multipliers and entirely generated in masked time. The resulting scheme created a vital balance between the computation capability and the data bandwidth that are required by the FFT algorithm, taking into account the VLIW architecture. Moreover, an important amount of input buffers can be freed since twiddle factors are no longer stored, nor referenced from memory, avoiding significant memory stalls. Since the recurrent computation of twiddle factors using multipliers induces a processing error, a tradeoff parameter between accuracy and speed is introduced.

Besides, our proposed implementation methodology takes into account the memory hierarchy, the memory banks, the cache size, the cache associativity, and its line size, in order to well adapt the FFT algorithm to a broad range of embedded VLIW processors. The bit reversal was efficiently designed to take advantage of the cache structure, and a mixed scheme was proposed for FFT/iFFT sizes not fitting the cache size.

VLIW assembly code of the FFT/iFFT was generated using a scheduling heuristic. The proposed FFT-specific modulo instruction scheduling algorithm is resource-constrained and register-sensitive that uses controlled backtracking. This heuristic re-orders the scheduling array to accelerate the backtracking search for the best schedule within the NP-complete state space. Our algorithm applies a strict optimization on internally used registers, so that generating twiddle factors of the new

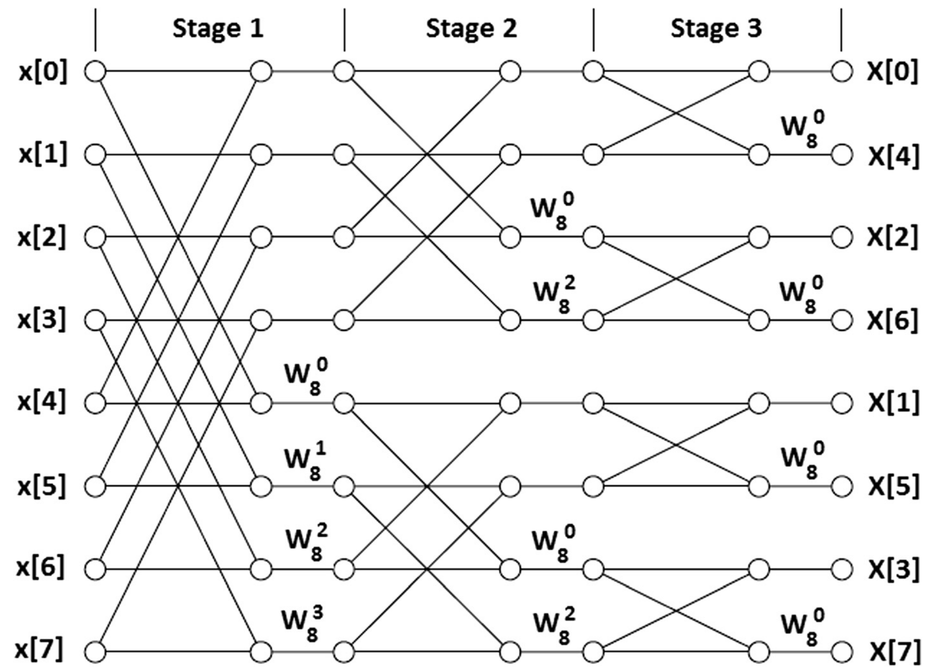


Fig. 2 8-point radix-2 DIF FFT

FFT scheme can be done effectively masked in parallel with other VLIW instructions.

The idea of recurrently generating twiddle factors during the FFT calculation is also discussed in our paper in [30]. In the present paper, we additionally propose a VLIW-generic recurrent FFT scheme and the related instruction scheduling generator.

In the following, a background on the FFT algorithm of interest is given in Section 2, we then present an overview on the VLIW DSP processors in Section 3, the new FFT strategy scheme is explained in Section 4, and the modulo-like scheduling of the suggested FFT is described in Section 5. Finally, experimental results are given in Section 6.

2 Background on the FFT algorithm of interest

Many factors other than the pure number of arithmetic operations must be considered for an efficient FFT implementation on a VLIW DSP, which can be derived from memory-induced stalls, regularity, and algorithm's projection on hardware VLIW architectures. Yet, one of the FFT algorithms that proved enough satisfaction on DSPs is the radix-4 FFT, mostly implemented by manufactures. This is mainly due to its relatively less access to memory (during $\log_4(N)$ stages vs. $\log_2(N)$ stages in a radix-2 scheme), additional to its regular, straightforward and less complex algorithm (compared for instance to split-radix FFTs, radix-8 and higher radix FFTs). The radix-4 FFT is usually used and mixed with a last radix-2 stage, enabling it to treat sizes that are power-of-2 and not only being limited to power-of-4 sizes.

We distinguish between two basic types of radix-4 algorithms: decimation-in-time (DIT) is the consequence of decomposing the input $x[n]$, and decimation-in-frequency (DIF) when decomposing the output $X[n]$. Both DIT and DIF present the same computational cost. In this paper, we will be only interested in DIF versions.

Building the radix-4 algorithm can be done starting from the DFT formula in (1) which can be rewritten in a decomposed form and consequently obtaining the DIF radix-4 butterfly description. An FFT computation is transformed into four smaller FFTs through a divide-and-conquer approach, making a radix-4 scheme with $\log_4(N)$ stages, each containing N/4 butterflies.

Through the given scheme, when the FFT input is in its natural order, the output will be arranged in the so-called digit-reversed order, which is defined depending on the used radix. Worthwhile to note that radix-2 FFT schemes present easier re-ordering process.

In 1996, He and Torkelson proposed in [31] a radix-4 variant, having the same computational complexity that they called a radix-2^2 FFT. Its main provided advantage is retaining the same butterfly structure as the radix-2 FFT and therefore preserving a bit-reversed order at the output. This is useful not only when re-ordering the output compared to 4-based digit-reversed re-ordering but also when mixing it with a last radix-2 stage, which in this case can be directly done, thanks to the instinct compatibility between radix-2 and radix-2^2 schemes. The new butterfly definition is given in Fig. 3 and the resulting algorithm's structure in Fig. 4.

The radix-2^2 FFT is adopted in this work and forms the basis of our new FFT scheme. Next, we describe the targeted VLIW family and the related state-of-art modulo scheduling.

3 VLIW DSP processors

3.1 Architecture overview

VLIW platforms allow several heterogeneous operations to be executed in parallel, due to multiple execution units in their architecture, achieving high computation capability (Fig. 5). In this case, the instruction size is increased depending on the number of units (usually 128 or 256 bits). Variable instruction sizes can be used as well to avoid additional NOP operations for code optimization. Although that each operation requires a number of cycles to execute, VLIW instructions can be fully pipelined; consequently, new operations can start in every cycle at every functional unit.

In VLIW processors, instructions are pipelined with an out-of-order execution, thus, iterations are initiated in parallel at a constant rate called the initiation interval (II) [19] (Fig. 6).

Among industrial VLIW platforms, we mention the Texas Instruments TMS320C6x, Texas Instruments 66AK2Hx, FreeScale StarCore MSC8x, ADI TigerSharc, and Infineon Carmel.

3.2 Background on the modulo scheduling

Modulo scheduling is a software pipelining technique, exploiting ILP to schedule multiple iterations of a loop in an overlapped form, initiating iterations at a constant rate called the initiation interval.

Modulo instruction scheduling schemes are mainly heuristic-based as finding an optimal solution is proven an NP-complete problem. Basic common operations of this technique include computing a lower bound of the initiation interval (denoted MII) which depends on provided resources and dependencies. Next, starting from this MII value, search for a valid schedule respecting hardware and graph dependency constraints. If no valid schedule could be found, increase the II value by 1 and search for a feasible schedule again. This last process is repeated until a solution is found. Higher performance is achieved for lower II; increasing II will reduce the

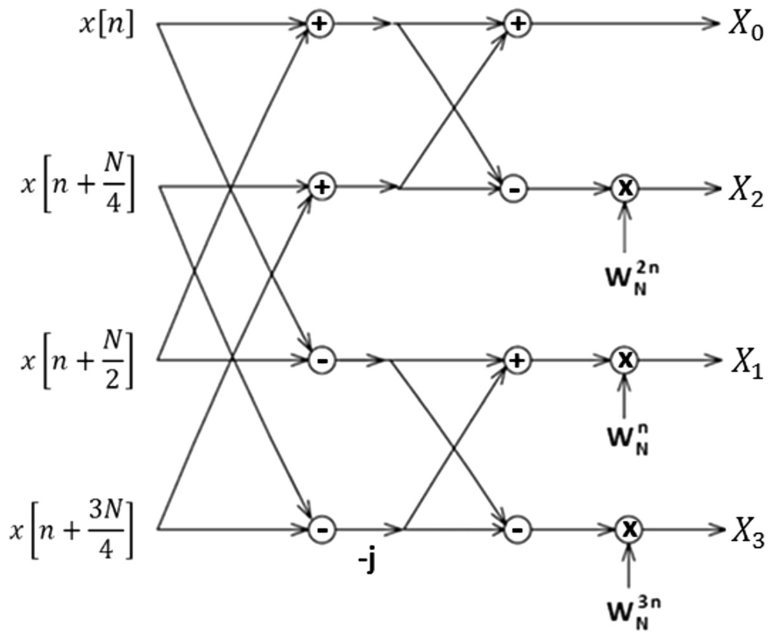

$x[n]$
$x\left[n+\frac{N}{4}\right]$
$x\left[n+\frac{N}{2}\right]$
$x\left[n+\frac{3N}{4}\right]$
-j
W_N^{2n}
W_N^{n}
W_N^{3n}
X_0
X_2
X_1
X_3

Fig. 3 Radix-2^2 butterfly

amount of used parallelism, however, this makes finding a valid schedule easier [32]. Main evoked techniques in the literature for modulo scheduling include iterative modulo scheduling (IMS) [19], which uses backtracking by scheduling and un-scheduling operations at different time slots searching for a better solution. Slack modulo scheduling [33] minimizes needed registers by reducing lifetimes of operands, using their scheduling freedom (or slack). Integrated register-sensitive iterative software pipelining (IRIS) [34] modifies the IMS technique to

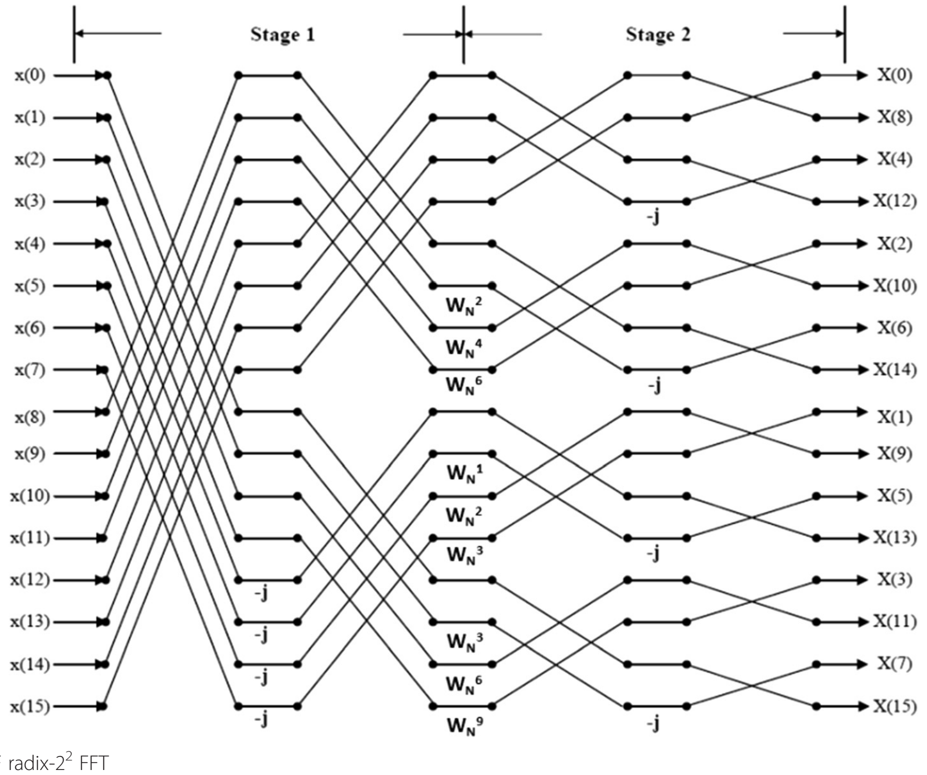

Stage 1
Stage 2
x(0) x(1) x(2) x(3) x(4) x(5) x(6) x(7) x(8) x(9) x(10) x(11) x(12) x(13) x(14) x(15)
-j -j -j -j
W_N^2 W_N^4 W_N^6 W_N^1 W_N^2 W_N^3 W_N^3 W_N^6 W_N^9
-j -j -j -j
X(0) X(8) X(4) X(12) X(2) X(10) X(6) X(14) X(1) X(9) X(5) X(13) X(3) X(11) X(7) X(15)

Fig. 4 16-point DIF radix-2^2 FFT

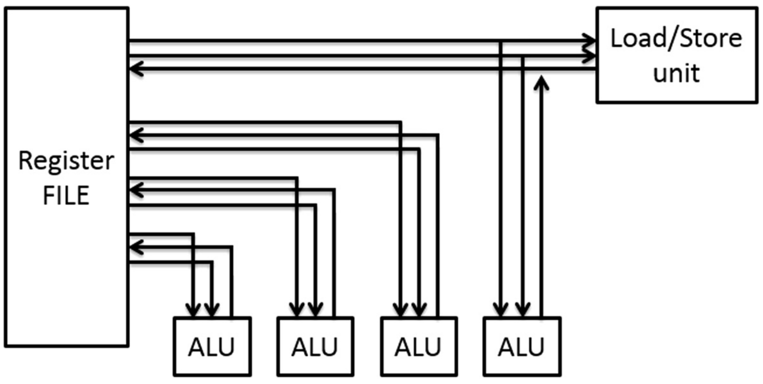


Fig. 5 Typical structure of a VLIW machine

minimize register requirements. Swing modulo scheduling (SMS) [35, 36] does not use backtracking but orders graph nodes guarantying effective schedules with low register pressure. The modulo scheduling with integrated register spilling (MIRS) in [37] suggests to integrate the possibility of storing data temporally out to memory (using spill code) when a schedule aims to exceed the number of available registers in the processor.

In general, the problem statement starts from a directed acyclic graph (DAG), with nodes representing operations of a loop, and edges for the intra- or inter-iteration dependencies between them. Those are valued at instructions' latencies. The wanted schedule must be functionally correct regarding data dependencies and hardware conflicts, minimizing both II and register usage, therefore reducing the execution time. Register pressure is a critical element to consider while searching for a schedule; a commonly used strategy is to minimize the lifetime of instruction's inputs/outputs.

Accordingly, modulo scheduling focuses on arranging instructions in a window of II slots called the kernel, when *m* iterations are overlapped within it, then *m-1* iterations must be separately done before and after entering the kernel; those are called prolog and epilog, respectively. In order to reduce the code expansion issue that is naturally required by modulo scheduling (typically for the prolog/epilog parts), hardware facilities for software pipelining are implemented in VLIW.

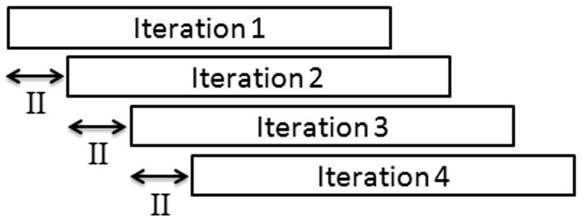


Fig. 6 Parallel initiation of iterations in VLIW cores

Next, our new FFT scheme is discussed for implementation possibilities on VLIW DSPs.

4 Our implementation methodology for the FFT on VLIW DSPs

4.1 A motivating example

The key idea behind the FFT scheme that we are proposing is to create a balance between the computation capability and the data bandwidth that are required by the FFT algorithm. In the following, we analyze the conventional FFT scheme regarding needed VLIW operations in a TI C66 device, in order to evaluate the default efficiency of resource usage.

The TI TMS320 C66 is a fixed and floating-point DSP. It contains eight cores, each working with a clock frequency of 1.0 to 1.25 GHz. A C66 CorePac core is able to execute up to eight separate instructions in parallel, thanks to the eight functional units in its VLIW structure as in Fig. 7. The maximum ability of one core is to execute 8 fixed/floating 32-bit multiplications per cycle using .M units, 8 fixed/floating 32-bit additions/subtractions per cycle using .L and .S units and loads/stores of 128-bit per cycle using .D units. The internal register files are composed of 64 32-bit registers [A31:A0 and B31:B0] [38].

As seen previously, the conventional radix-2^2 FFT algorithm needs $\log_4(N)$ stages (assuming N a power-of-4); and within each stage, a number of N/4 radix-2^2 butterflies are computed. Besides, there is a need of 8 loads/stores for each butterfly's legs, in addition to 3 load operations of twiddle factors as shown in Fig. 3; making a total of 11 loads/stores per butterfly. Moreover, eight complex additions/subtractions, three complex multiplications, and a special $(-j)$ multiplication are required to complete the processing on a single butterfly.

Let us denote n_d the number of loaded/stored data in bytes during the whole FFT algorithm, n_m the total number of required real multiplications, n_a the total number of required real additions/subtractions, and n_j the total

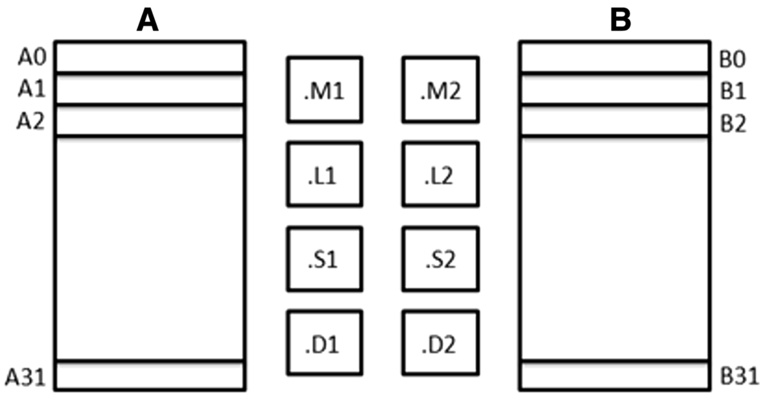


Fig. 7 The 8-way synoptic structure with 8 VLIW units (denoted as .M1/2, .L1/2, .S1/2 and .D1/2) and register files (A31:A0 and B31:B0) of a C66 core

number of ($-j$) multiplications. Then, the conventional algorithm needs the (n_d, n_m, n_a, n_j) expressed in (2) (always assuming N a power-of-4, denoting $B_N = N\log_4(N)/4$ as the total number of radix-4 butterflies). The n_d formula is scaled by a factor of 8 since the used samples are single-precision floating point, hence, coded in 8 bytes for both real and imaginary parts. The n_m formula takes into consideration that each complex multiplication is translated into 4 real multiplications, and the n_a formula sums the additions/subtractions that are needed in both complex additions/subtractions {16} and in complex multiplications {6}.

$$\begin{cases} n_d = 88\,B_N = 22\,Nlog_4N \\ n_m = 12\,B_N = 3\,Nlog_4N \\ n_a = 22\,B_N = \dfrac{11}{2}\,Nlog_4N \\ n_j = B_N = \dfrac{N}{4}log_4N \end{cases} \tag{2}$$

A VLIW core can especially issue loads/stores, multiplications, additions/subtractions, and ($-j$) multiplications in parallel. In the C66 core case, load/store capacity is 16 bytes per cycle using .D units (denoted next by p_d). Eight real floating-point single-precision multiplications can be done per cycle (p_m) using *.M* units and eight real floating-point single-precision additions/subtractions are achievable per cycle (p_a) using both .L and .S units. Finally, 2 multiplications by ($-j$) per cycle (p_j) using .L and .S units as well, those last are simplified into combinations of move and sign-change operations between real and imaginary parts.

According to these VLIW core capacities, the maximum peak performance in terms of clock cycles for the whole conventional FFT on device is equal to:

$$\mathrm{MAX}\left(\frac{n_d}{p_d}, \frac{n_m}{p_m}, \frac{n_a}{p_a} + \frac{n_j}{p_j}\right) = \frac{11}{8} N\,log_4(N)$$

The minimal need for each set of operations on the given C66 VLIW core is shown in Fig. 8. The most optimized conventional way for implementing this algorithm at instruction level leads to the kernel form presented in Fig. 9. It shows butterfly's instructions mapping on the eight VLIW functional units (.D1, .L1, .S1 …). Since each

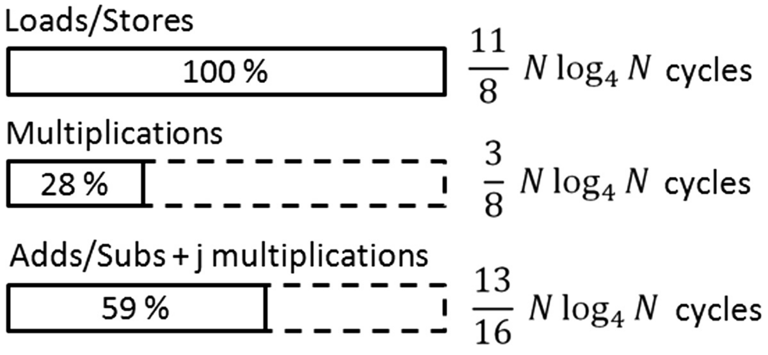


Fig. 8 Minimal need of the conventional FFT algorithm in terms of clock cycles on C66

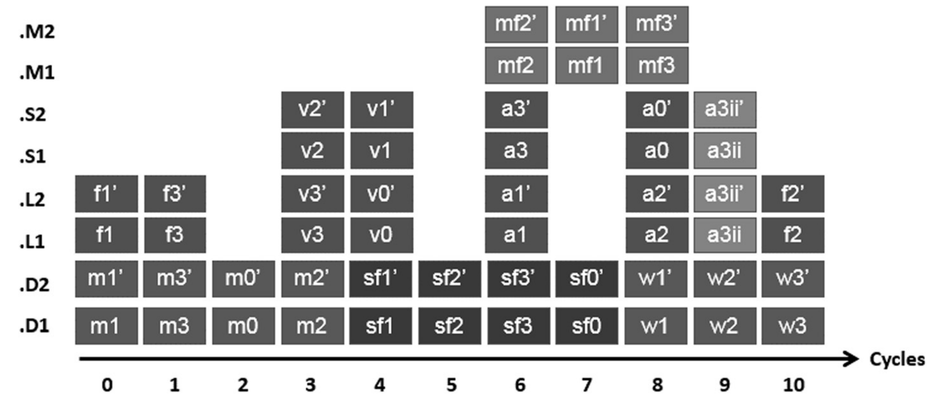


Fig. 9 The VLIW kernel form of the conventional FFT implementation on a C66 core (II = 11)

C66 core contains two side banks (*A* and *B*), we processed symmetrically one butterfly per side.

In Fig. 9, loads of the four butterfly's legs are marked by m0, m1, m2, and m3. Stores are represented by the entities sf0, sf1, sf2, and sf3. Additions and subtractions are referred as a0, a1, a2, a3, v0, v1, v2, and v3. A multiplication by $(-j)$ is translated into two operations on .L/.S units: a3ii (a3i1, a3i2). In addition, multiplications are marked by mf1, mf2, and mf3; and since we are processing complex data (with real and imaginary parts), extra additions are needed to complete the complex multiplication operation: f1, f2, f3. Finally, the symbols w1, w2, and w3 represent loads from memory of needed twiddle factors (W_N^k).

Hence, there is indeed an unbalance between the required computation capability and the load/store bandwidth for this VLIW case, making excessive loads/stores vs. lower use of computation units.

Our new method changes the structure of the algorithm to fit the VLIW hardware, creating a balance in resource usage, and therefore minimizing the overall clock cycles. In an attempt to reduce the load/store pressure, we suggest not to load twiddle factors but to generate them internally instead. This idea uses the fact that the twiddle factors in the *n*-indexed butterfly that are (W_N^{2n}, W_N^n, W_N^{3n}) can be deduced from the $(n-1)$-indexed butterfly according to these formulas: ($W_N^{2n} = W_N^{2(n-1)}W_N^2$, $W_N^n = W_N^{n-1}W_N$, $W_N^{3n} = W_N^{3(n-1)}W_N^3$). This trades loads/stores for multiplications/additions and makes an FFT scheme with butterflies that are dependent on each other; therefore, we cannot start the processing of the *n*-indexed butterfly if the $(n-1)$-indexed butterfly did not yet compute its twiddle factors.

We decide to process one butterfly per C66 core side bank as well, grouping two butterflies within a single larger iteration. Therefore, this makes an II large enough to wait for the generation of needed twiddle factors by subsequent pipeline stages (7 cycles are required on the TI C66 device in order to complete a floating-point complex multiplication).

This new scheme reduces the loads/stores in exchange with arithmetic operation increase (Fig. 10). Indeed, the number of needed cycles for loads/stores becomes $N \log_4(N)$ as twiddle factors are no more loaded, while the numbers of multiplications and additions/subtractions are increased to $3N\log_4(N)/4$ cycles and $N\log_4(N)$ cycles, respectively.

Our modulo scheduling heuristic that will be detailed in the next sections was able to generate the aggressively optimized schedule of Fig. 11 for the new FFT scheme. The obtained Initiation Interval (II) is 8 cycles instead of 11, fitting a limited core register set, despite adding extra arithmetic operations for twiddle factor generation. This enhances the raw needed time for the FFT by 27 % (excluding memory stall gains) over the conventional implementation with twiddle factors. Moreover, this scheme requires 0 % references to twiddle factors and 0 % space for their memory storage as well.

4.2 Proposed FFT implementation methodology

Our implemented methodology allows better core-level resource balance, exploiting the fact that the twiddle factors can be calculated recurrently using multipliers during the execution. The resulting scheme, regarding a VLIW architecture, created a vital balance between the computation capability and the data bandwidth that are required by the FFT algorithm. Besides, it takes into account the memory hierarchy, the memory banks, the cache size, the cache associativity, and its line size, in order to well adapt the FFT algorithm to a broad range of embedded VLIW processors. The bit reversal was efficiently designed to take advantage of the cache structure, and a mixed scheme was proposed for FFT/iFFT sizes not fitting the cache size.

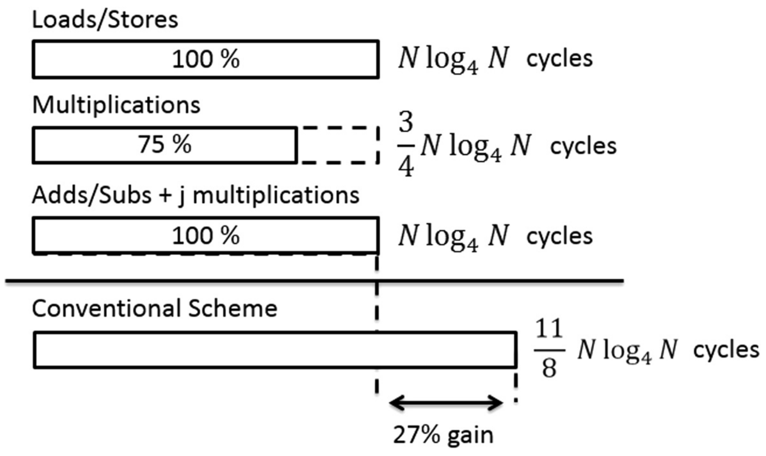


Fig. 10 Bandwidth and computation power's comparison for the new scheme on C66 cores

4.2.1 A recurrent FFT scheme without memory references of twiddle factors

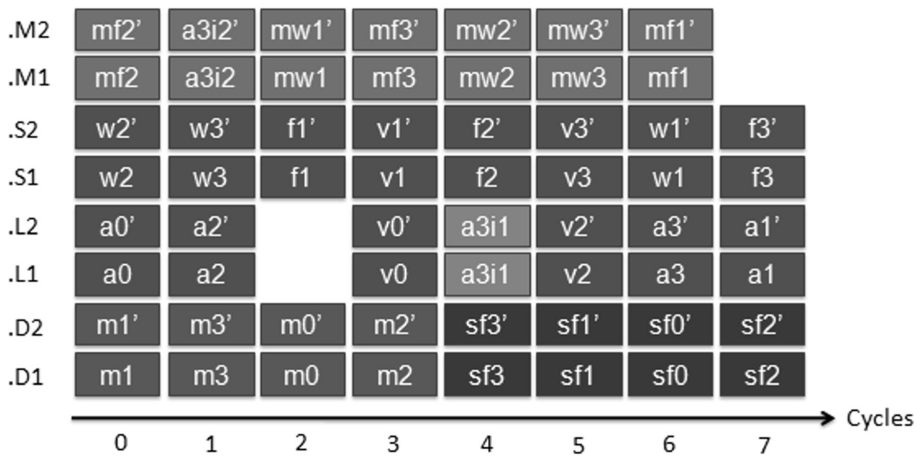
The proposed FFT scheme generates twiddle factors using multipliers instead of loading them from a pre-computed array. Those are recurrently computed from previously handled butterflies. Therefore, the processing of the n-indexed butterfly is time-constrained by the $(n - 1)$-indexed butterfly.

Let us denote t_w the latency time in terms of cycles that is needed to compute a twiddle factor from a previously calculated one; consequently, if an iteration processes one butterfly, then the II must be greater than or equal to t_w, waiting the required time to generate twiddle factors for the next iteration.

For maximum FFT performance, the initiation interval must be minimized, expressed for our new FFT as MII = MAX(RCPB {required cycles per butterfly} for loads/stores, RCPB for multiplications, RCPB for adds/subs and j multiplications, t_w).

In order to mask the effect of t_w on II, we unroll a number of U successive iterations into a single large one, reducing dependencies to between groups of merged iterations. The new MII expression becomes MII = MAX($U \times$ RCPB for loads/stores, $U \times$ RCPB for multiplications, $U \times$ RCPB for adds/subs and j multiplications, t_w).

In the earlier equation, $3U$ twiddle factors per group are deduced from $3U$ previously calculated ones, which reduces dependencies toward the constant delay t_w. The minimum value of U minimizing MII (having MII > t_w) will be denoted U_m next. During each n-indexed group of U_m butterflies, there is a need of $3U_m$ twiddle factors which are computed from those in the $(n - 1)$-indexed group as in (3) (denoting twiddle factors by $\gamma_{n,k}$). On the



	0	1	2	3	4	5	6	7
.M2	mf2'	a3i2'	mw1'	mf3'	mw2'	mw3'	mf1'	
.M1	mf2	a3i2	mw1	mf3	mw2	mw3	mf1	
.S2	w2'	w3'	f1'	v1'	f2'	v3'	w1'	f3'
.S1	w2	w3	f1	v1	f2	v3	w1	f3
.L2	a0'	a2'		v0'	a3i1	v2'	a3'	a1'
.L1	a0	a2		v0	a3i1	v2	a3	a1
.D2	m1'	m3'	m0'	m2'	sf3'	sf1'	sf0'	sf2'
.D1	m1	m3	m0	m2	sf3	sf1	sf0	sf2

Fig. 11 The best FFT schedule on C66 returned by the heuristic (a kernel of II = 8)

other hand, treating many butterflies simultaneously per iteration requires the usage of more VLIW core registers.

$$\begin{cases} \gamma^2_{n,k} = \gamma^2_{n-1,k}\, W_N^{2U_m} \\ \gamma_{n,k} = \gamma_{n-1,k}\, W_N^{U_m} \\ \gamma^3_{n,k} = \gamma^3_{n-1,k}\, W_N^{3U_m} \end{cases} \tag{3}$$

For $0 \le k < U_m$ and $1 \le n < \frac{N}{4U_m}$ Denoting $\gamma_{n,k} = W_N^{nU_m+k}$

The resulting structure of a group of U_m butterflies composing a single larger iteration is shown in Fig. 12. The minimum II will be then expressed as follows:

$$\mathrm{MII} = \mathrm{MAX}\Big(U_m n'_d/(p_d B_N), U_m n'_m/(p_m B_N), U_m\Big(n'_a/p_a + n'_j/p_j\Big)/B_N\Big)$$

with (n'_d, n'_m, n'_a, n'_j) are the new needs in terms of arithmetic and load/store operations expressed as in (4).

$$\begin{cases} n'_d = 64\, B_N = 16\, N log_4 N \\ n'_m = 24\, B_N = 6\, N log_4 N \\ n'_a = 28\, B_N = 7\, N log_4 N \\ n'_j = B_N = \dfrac{N}{4} log_4 N \end{cases} \tag{4}$$

Consequently, this new scheme reduces memory accesses by 27 %, making an implementation advantage on a broad range of architectures as most FFT algorithms use memory extensively. In addition to that, it gives an opportunity to use the non-exploited VLIW units for a possibly masked generation of twiddle factors. Besides, since it is not an obvious task to generate an efficient pipelined schedule (having II = MII) with respect to hardware constraints and available core registers, we suggest in later sections an aggressive FFT-specific scheduling heuristic.

This scheme requires 0 % references to twiddle factors and 0 % space for their memory storage as well, making significant gains on related memory latencies.

The key parameters of our scheme are the VLIW core features (p_d, p_m, p_a, p_j, t_w). By computing the MII_1 when using an FFT scheme with loaded twiddle factors (using n_d, n_m, n_a, and n_j, expressed in Eq. (2)), and MII_2 when using an FFT scheme with recurrent computation of twiddle factors (using n'_d, n'_m, n'_a, and n'_j, expressed in Eq. (4)), then if $\mathrm{MII}_1 < \mathrm{MII}_2$, the provided VLIW core would not be applicable for the proposed scheme; otherwise, the minimal gain is expressed as $100(\mathrm{MII}_1 - \mathrm{MII}_2)/\mathrm{MII}_1$.

4.2.2 Setup code integration within the pipelined flow

The previous section described low-level instruction mapping of the most inner loop of the new FFT scheme. In order to complete the FFT/iFFT implementation, intermediate setup iterations (representing outer loops) must be injected into the pipelined flow of iterations. The straightforward way is to completely drain the last iterations of the inner loop, executing the setup code (constants reset, pointers updates …), and then resuming the pipelined execution; this turns to be time-consuming due to the time needed for the prolog/epilog parts. Indeed, if the dynamic length (DL) denotes the number of cycles that are needed by an inner loop iteration for its processing, then the whole FFT will at least require $B_N\mathrm{MII}/Um$ + (DL-II)(N/48 – 1/3) cycles (assuming N a power-of-4). The integer expression (N/48 – 1/3) counts the number of interrupts that must be done to the inner loop kernel, assuming that the last two FFT stages can be especially treated and done without setup code merging. For $N = 4k$ on the C66 for example, we can see that the setup code interruptions represent 7 % of the main processing.

VLIW architectures can support the merging of setup codes into the pipelined iterations (the case of C66), making it possible to add a customized iteration in concurrence with others without draining the process. Therefore, we can insert an additional II cycle iteration, setting up changes required by outer loops to begin the next sequence of inner loop iterations; needing only $B_N\mathrm{MII}/Um$ + II(N/48 – 1/3) cycles on the whole FFT/iFFT routine. This enhances the efficiency, representing only 2.7 % of the main processing when $N = 4k$ on C66 cores.

4.2.3 Cache-efficient bit reversal computation

The FFT naturally returns an N-sized output with a bit-reversed order, post re-ordering data is necessary. Bit reversing an element at a 0-based index k consists of replacing it into the position index m, such that m is obtained by reversing $\log_2(N)$ bits of the binary form of k. Processors usually implement hardware bit reversal of a 32-bit integer, hence, the wanted function can be obtained by left-shifting an index 32-$\log_2(N)$ times and bit-reversing the whole word afterwards.

In order to increase computation efficiency, one can integrate the re-ordering step into the last stage of the FFT/iFFT, rather than creating a separate routine. One encountered difficulty in bit reversal is accessing scattered positions of memory, causing many memory stalls. Indeed, commonly used architectures of L1D cache are to divide it into several banks, such that simultaneous accesses to distinct banks are possible, while many concurrent accesses to the same bank induce latencies. In processors' architectures that allow multiple data to be accessed in a cycle, the L1D cache level is divided into banks that are defined by lower bits of an address (AMD's processors, TI C6x DSPs, …). In the C66 CorePac architecture for example, there is 8 L1D banks such that the address range [$4b + 32k$; $4(b + 1) + 32k$

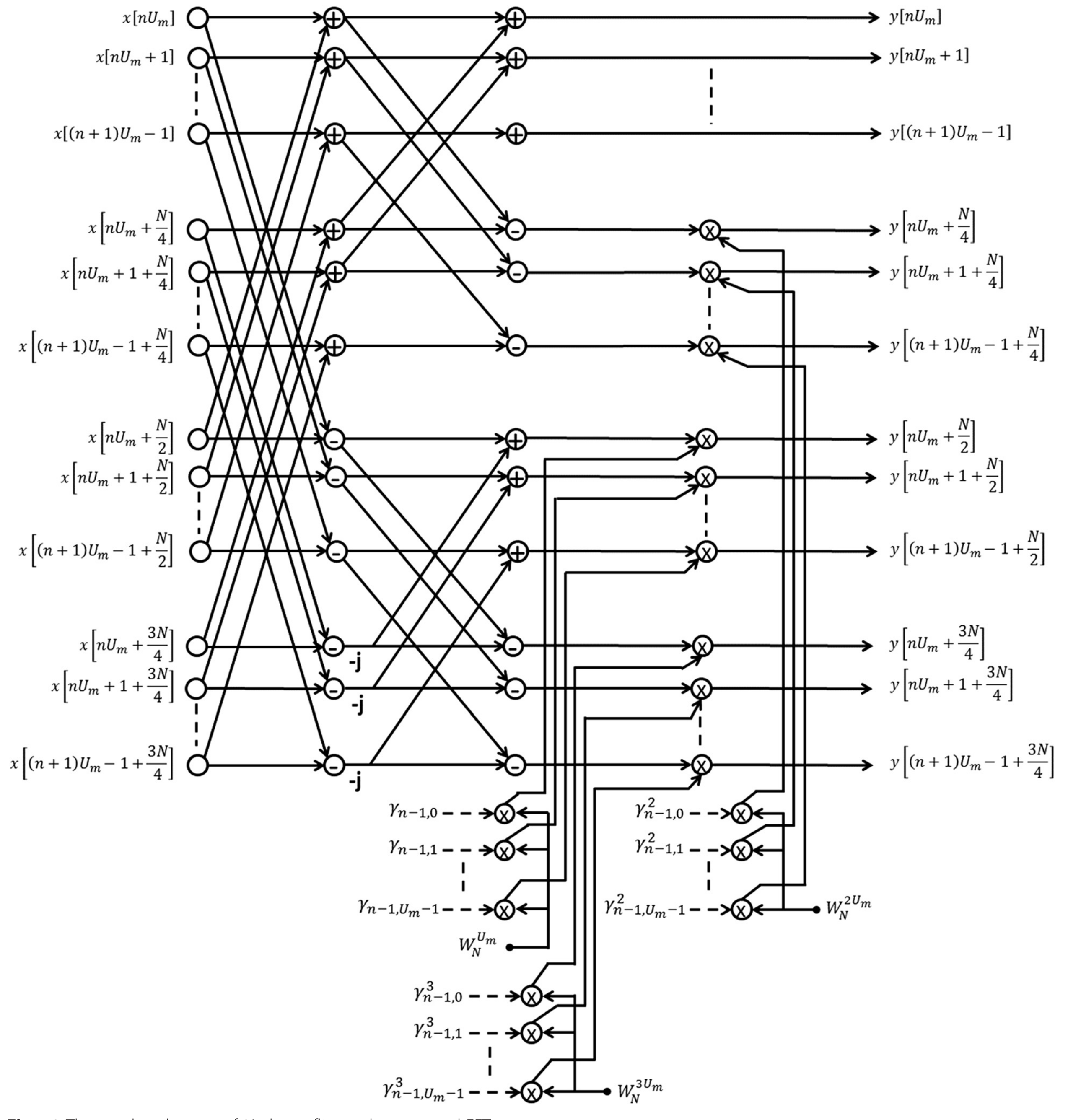

$x[nU_m]$
$x[nU_m+1]$
$x[(n+1)U_m-1]$
$x\left[nU_m+\frac{N}{4}\right]$
$x\left[nU_m+1+\frac{N}{4}\right]$
$x\left[(n+1)U_m-1+\frac{N}{4}\right]$
$x\left[nU_m+\frac{N}{2}\right]$
$x\left[nU_m+1+\frac{N}{2}\right]$
$x\left[(n+1)U_m-1+\frac{N}{2}\right]$
$x\left[nU_m+\frac{3N}{4}\right]$
$x\left[nU_m+1+\frac{3N}{4}\right]$
$x\left[(n+1)U_m-1+\frac{3N}{4}\right]$
-j
-j
-j
$y[nU_m]$
$y[nU_m+1]$
$y[(n+1)U_m-1]$
$y\left[nU_m+\frac{N}{4}\right]$
$y\left[nU_m+1+\frac{N}{4}\right]$
$y\left[(n+1)U_m-1+\frac{N}{4}\right]$
$y\left[nU_m+\frac{N}{2}\right]$
$y\left[nU_m+1+\frac{N}{2}\right]$
$y\left[(n+1)U_m-1+\frac{N}{2}\right]$
$y\left[nU_m+\frac{3N}{4}\right]$
$y\left[nU_m+1+\frac{3N}{4}\right]$
$y\left[(n+1)U_m-1+\frac{3N}{4}\right]$
$\gamma_{n-1,0}$
$\gamma_{n-1,1}$
γ_{n-1,U_m-1}
$W_N^{U_m}$
$\gamma^2_{n-1,0}$
$\gamma^2_{n-1,1}$
γ^2_{n-1,U_m-1}
$W_N^{2U_m}$
$\gamma^3_{n-1,0}$
$\gamma^3_{n-1,1}$
γ^3_{n-1,U_m-1}
$W_N^{3U_m}$

Fig. 12 The *n*-indexed group of U_m butterflies in the proposed FFT structure

[(where *k* and *b* are integers) is linked with the bank number *b* ($b \in [0,7]$).

It turns out that store indexes related to first butterflies (0, *N*/2, *N*/4, …) all usually belong to the same memory bank (as long as *N* gets high values); consequently, 2 parallel stores in a constructed kernel will likely target different addresses from the same bank, inducing stalls. Besides, it is recommended to access consecutive addresses, to be possibly merged into larger accesses by subsequent data paths.

The straightforward implementation provides no successive stores, neither avoiding bank conflicts. A possible enhancement could be achieved by trying to bit-reverse input indexes on the last FFT stage instead of output ones. Resulting structure of this FFT/iFFT stage is described below (processing four butterflies per iteration):

```
i = 0;
loop N/16 times
br = bit_reverse(i);
load_legs(br, br + 1, br + 2, br + 3);
load_legs(N/2 + br, N/2 + br + 1, N/2 + br + 2, N/2 + br + 3);
load_legs(N/4 + br, N/4 + br + 1, N/4 + br + 2, N/4 + br + 3);
load_legs(3 N/4 + br, 3 N/4 + br + 1, 3 N/4 + br + 2, 3 N/4 + br + 3);
process_butterflies();
store_legs(i, i + N/2, i + N/4, i + 3 N/4);
store_legs(i + 1, i + N/2 + 1, i + N/4 + 1, i + 3 N/4 + 1);
store_legs(i + 2, i + N/2 + 2, i + N/4 + 2, i + 3 N/4 + 2);
store_legs(i + 3, i + N/2 + 3, i + N/4 + 3, i + 3 4 + 3);
i = i + 4;
end loop
```

Doing so, we can always issue parallel stores targeting different memory banks and then avoiding bank conflicts (for example, data (i) at a specific bank in parallel with data ($i+1$) at another bank). Furthermore, 4 consecutive store accesses are now possible. In this case, butterflies are processed in an order that provides the maximum of consecutive stores.

During the FFT, in-place computation on an input buffer is performed until the last stage, where stored data are put into the output buffer. Processing 2, 4, or more butterflies per iteration increases register pressure; we have applied the same scheduling heuristic that will be described later to find a feasible implementation with advanced constraints on data accesses. Found schedule has II = MII with all constraints verified ensuring consecutive stores and avoiding bank conflicts. Obtained performance of the FFT routine with bit reversal was similar to an FFT without bit reversal.

For FFT sizes that are power-of-2 and not power-of-4, an additional radix-2 stage is added; that is where the bit reversal is merged in the same manner.

4.2.4 Adapting the FFT to the cache associativity

When the FFT size is greater than the allocated cache size, the considered radix-2^2 scheme may present some inefficiency toward the L1D cache. The L1D cache is composed of a number of cache lines (usually of 64 bytes), used to store external data prior to their use by the CPU. A cache miss occurs when the requested data is not yet available into the cache; in this case, the CPU is stalled waiting for a cache line to be updated. Many cache structures were used in CPU architectures: direct-mapped cache associates each address of the external memory with a unique position into the cache (therefore with one cache line); as a result, two addresses that are separated by a multiple of the cache size could not survive into the cache at the same time. An advanced mechanism is called the set-associative cache, where the cache memory is divided into a number of p ways, such that when a cache miss occurs, data is transferred to the way whose cache line is the least recently used (LRU); consequently, there are p unique locations into the cache for every address. The main advantage for increasing the associativity is to let non-contiguous data survive into cache lines without overwriting each other (without cache thrashes).

A radix-2^2 butterfly loads their legs from the indexes: 0, $N/4$, $N/2$, and $3N/4$. All this data should exist in the cache at the same time. If the L1D cache is 2-way set-associative and denoting L1D_S the allocated cache size in bytes, no cache thrash would happen if N is less than or equal to L1D_S/8. Otherwise, cache lines for indexes (0 and $N/2$) or ($N/4$ and $3\,0/4$) will overwrite each other continuously, decreasing then the cache efficiency. A solution to this consists of applying radix-2 FFTs for larger sizes, until the size (L1D_S/8) where radix-2^2 can be used without cache thrashes. Indeed, while radix-2 FFTs only access elements at indexes like (0 and $N/2$), no cache thrash would occur no matter how large N is; as long as the cache is 2-way set-associative.

A radix-2 FFT without references of twiddle factors was similarly built and generated using our scheduling heuristic leading to a schedule of II = MII (merging 4 radix-2 butterflies in a single iteration).

Denoting the cache associativity parameter by CACHE_A, the pseudo-code of our adapted FFT scheme regarding cache is written below:

```
function fft_cache()
begin
step = N;
while (step > L1D_S/8 and CACHE_A < 4) loop
for (k = 0;k < N/step)
fft_radix2_stage(input + k*step, step);//radix-2 fft
step = step/2;
end loop
for (k = 0;k < N/step)
fft(input + k*step, step);//radix-$2^2$ fft
bit_reversal(input, output);
end
```

A bit reversal routine was designed separately in this case and cache-optimized; the II(MII) was extended in order to optimally (fully) treat 4 cache lines in each iteration.

Using a radix-2 FFT for first stages is making a slight drop on the overall efficiency. In fact, a full-radix-2 scheme requires more time than a full-radix-2^2 scheme (on C66 cores, it needs $N\log_2(N)$ cycles at peak performance instead of $N\log_4(\mathrm{N})$). Even so, its gain is far dominant for large FFTs avoiding cache thrashes. For example, the 16*k*-FFT performance on C66 cores using a full-radix-2^2 scheme is 367,890 cycles; it decreased to 245,206 cycles (33 % gain) using the scheme-avoiding cache thrashing.

The inverse FFT is the same as a FFT, except the fact that it uses conjugated twiddle factors and ($1/N$) extra multiplications added to the last stage. These modifications can be performed without decreasing performance, by exploiting the ILP feature of VLIW processors.

4.2.5 FFT scheme accuracy

A possible side effect of the proposed implementation is a slightly reduced precision. Indeed, internally computed twiddle factors in a recurrent fashion are less accurate than those loaded pre-computed using trigonometric functions. We introduce in Fig. 13 a tradeoff parameter (tradeoff_factor) between accuracy and speed. The tradeoff scheme injects more pre-computed twiddle factors within the FFT flow instead of using only one, which reduces error accumulation effects. However, since the pipeline is regularly stopped to process more pre-computed twiddle factors, the speed performance slightly drops.

The key idea of the algorithm in Fig. 13 is to use more than one pre-computed twiddle factor per FFT stage in order to limit the error propagation. Indeed, if the tradeoff_factor is 0 then only 1 twiddle factor will be used to feed the whole FFT process. Otherwise, 2^(tradeoff_factor)/2 pre-computed twiddle factors will be used per each FFT stage. We will denote next the tradeoff_factor by T.

The calculation error of twiddle factors using the repeated multiplication algorithm grows as $O(N)$ as shown in [39]. Therefore, using our tradeoff method, the twiddle factors accuracy would be expressed as follows:

$$\left|\omega_N^k - \widehat{\omega}_N^k\right| \le \left(4\frac{\sqrt{3}}{3} + \frac{\sqrt{2}}{2}\right)\left(\frac{N}{2^{T+1}\,U_m}\right)u, k < N$$

where $\widehat{\omega}^{Nk}$ is the approximated twiddle factor value and u is the unit roundoff. For IEEE-754 single-precision floating point, the unit roundoff is equal to $u = 5.96 \times 10^{-8}$.

For increasing values of *T*, the method's accuracy increases, but also the FFT time increases according to the following formula:

```
function fft_tradeoff (tradeoff_factor)
begin
   if (tradeoff_factor==0)
   begin
      fft(input, output, N, w[1]);
      return;
   end
   ss=N/ (2^(tradeoff_factor-1));
   step=N;   j_s=1;
   while (step>4) loop
       ss_cnt=step/ss;
       for (k=0;k<N/step)
       begin
           j=j_s;
           if (ss_cnt>1)
           begin
               for (s=0;s<ss_cnt)
                   fft_radix2p2_stage(input+k*step+s*ss, ss, w[j++]);
           else
               fft_no_last_stage(input+k*step, step, w[j:log4(step)]);
               step=16;
           end
       end
       step=step/4;   j_s=j;
   end loop
   last_fft_stage(input, output, step);
end
```

Fig. 13 Pseudo-code for an FFT scheme with an accuracy-tradeoff parameter

$$\frac{B_N MII}{U_m} + II\left(\frac{N}{48} - \frac{1}{3}\right) + 2^{T-1}(DL - II)log_4(N)$$

Where DL denotes the dynamic length of an inner loop iteration. The T parameter creates then a tradeoff factor between accuracy and speed.

5 Instruction scheduling heuristic for FFT/iFFT implementations on VLIW

5.1 Introduction

According to the proposed FFT scheme, a group of U_m butterflies must be processed during a single iteration. Furthermore, extra operations are added to compute the needed twiddle factors in the masked time; these make an iteration with largely increased operations to be scheduled during II = MII within a limited core register set. The data dependence graph of this inner loop FFT iteration is shown in Fig. 14; each node of the graph (among $30U_m$ nodes) can be executed in 1 functional unit of the VLIW core.

In a TI C66 device for instance, and as we aspire the schedule to be done for II = MII = 8, the kernel can be able then to execute up to 64 nodes (due to the 8-way VLIW architecture). It leads to a functional unit pressure of 60 per 64 possible slots (94 % of unit pressure); this shows the difficulty class of the actual scheduling problem, in the presence of a limited register set.

The new FFT scheme merges U_m totally independent butterflies in a single iteration, making it possible to symmetrically divide the computation on processors with symmetric core-bank structure (ADI TigerShark, TI C66 [Fig. 7], ...). This will have the effect to reduce the problem size by half (on $U_m/2$ butterflies), and avoid core-bank communication which is therefore usually limited with many other constraints.

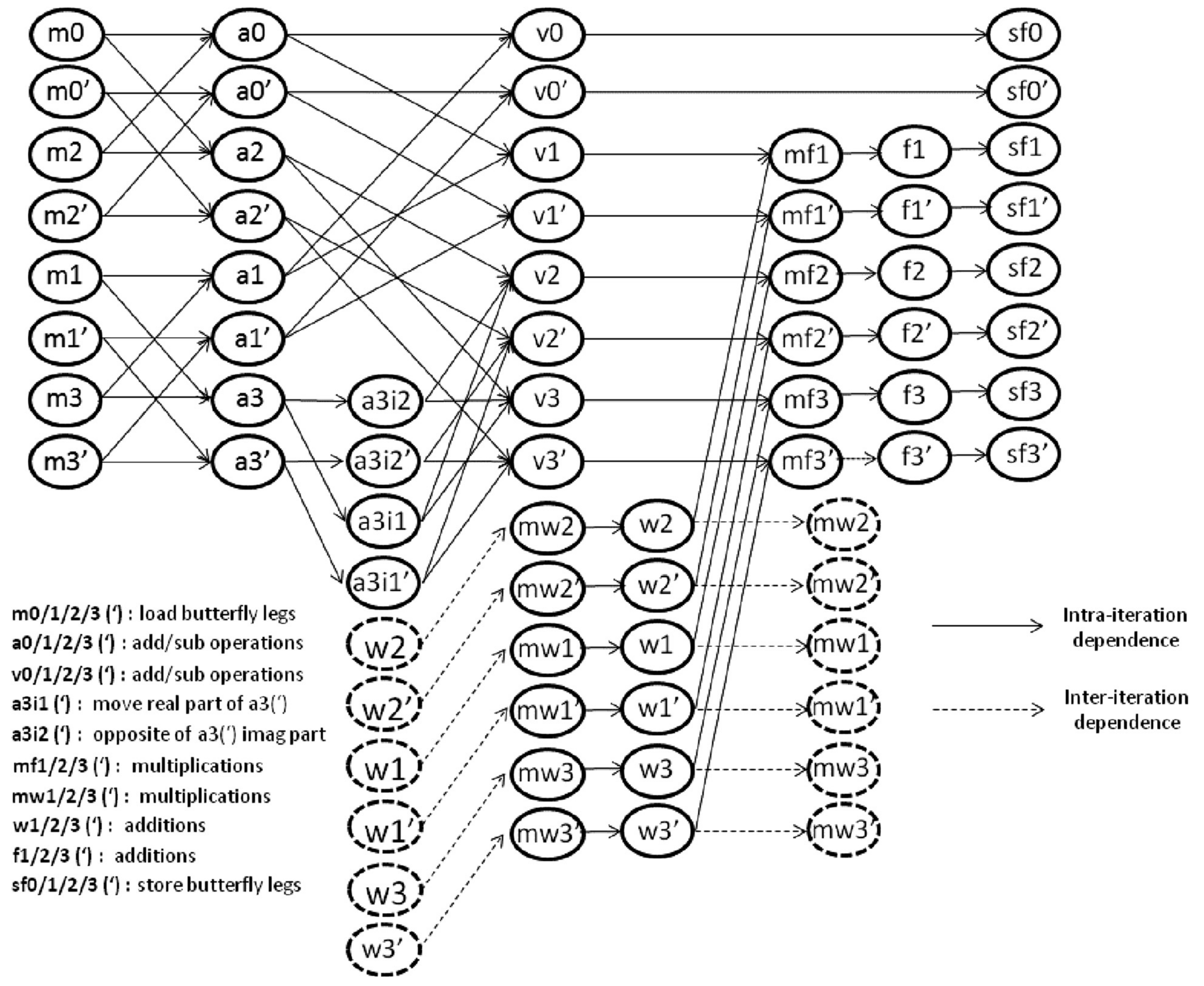

m0
m0′
m2
m2′
m1
m1′
m3
m3′
a0
a0′
a2
a2′
a1
a1′
a3
a3′
a3i2
a3i2′
a3i1
a3i1′
w2
w2′
w1
w1′
w3
w3′
v0
v0′
v1
v1′
v2
v2′
v3
v3′
mw2
mw2′
mw1
mw1′
mw3
mw3′
mf1
mf1′
mf2
mf2′
mf3
mf3′
f1
f1′
f2
f2′
f3
f3′
sf0
sf0′
sf1
sf1′
sf2
sf2′
sf3
sf3′
m0/1/2/3 (′) : load butterfly legs
a0/1/2/3 (′) : add/sub operations
v0/1/2/3 (′) : add/sub operations
a3i1 (′) : move real part of a3(′)
a3i2 (′) : opposite of a3(′) imag part
mf1/2/3 (′) : multiplications
mw1/2/3 (′) : multiplications
w1/2/3 (′) : additions
f1/2/3 (′) : additions
sf0/1/2/3 (′) : store butterfly legs
Intra-iteration dependence
Inter-iteration dependence

Fig. 14 Iteration DAG of the new FFT scheme (case of $U_m = 2$)

Each node of the graph is associated with a specific assembly instruction, requiring input/output operands with precise sizes, and a list of possible functional units for execution (an example of some TI C66 instructions in Table 1).

At the core-register level, the FFT algorithm will need to allocate a number of registers exclusively to store data pointers, constants, counters, and twiddle factors. First, 1 register is needed to store a counter on the iteration loop; 4 registers per butterfly to contain pointers on input and output butterfly legs; 1 register for the FFT stride; 3 registers per butterfly for jump offsets; 1 register for a pointer on the final output buffer; 1 register as a stack pointer (where some core-registers are spilled). Besides, 3 register pairs are exclusively needed for $\left(W_N^{U_m}, W_N^{2U_m}, W_N^{3U_m}\right)$, and $3Um$ twiddle factors per butterfly must be allocated as well. Left registers must be used for operand allocation of the entire FFT/iFFT DAG. Our scheduling algorithm must take into account this limiting register constraint.

One of the most efficient scheduling heuristic is SMS as evaluated in [32, 35, 36]. Applying SMS on our FFT problem in the TI C66 core, it produced a schedule of II = MII with a minimum register usage of 20 per core side (40 needed registers), which is greater than the available registers for the DAG allocation, meaning that this is not an implementable schedule. Our scheduling technique aims to find a valid schedule with II = MII and a minimum register requirement in a reasonable time.

5.2 A proposed modulo scheduling algorithm

The new scheduling algorithm starts with ordering the graph nodes for a one core side into a 1-dimensional array, such that if a node is j-indexed into this array, all of its predecessors must be indexed less than j. This is the case as the scheduling algorithm that will be presented next uses this generated array-order to schedule/un-schedule graph nodes in a backtracking fashion and computes the best starting time of each node (regarding register lifetime) based on all of its already scheduled predecessors. The scheduling order is critical and our algorithm uses a special ordering of graph nodes, which will be presented later.

Table 1 Instructions' features of some TI C66 operations

Instruction	Operand size (op1, op2, dst) [in number of core registers]	Delay slot latency (DS)	Possible execution units
DADDSP	(2, 2, 2)	2	L, S
DSUBSP	(2, 2, 2)	2	L, S
CMPYSP	(2, 2, 4)	3	M
STDW	(2, 0, 1)	0	D
LDDW	(1, 0, 2)	4	D
ADD	(1, 1, 1)	0	L, S, D

For each node on the ordered list, we define two parameters that are node possible start time (NPST) and node end time (NET), which describe the freedom or possible slots in the actual schedule; they are defined according to (5). We define as well the node start time (NST) as the effective start time, varying between NPST and NET, and pred$_i$ being the ith predecessor of a node. Special freedom ranges were attributed for root nodes not having any predecessor and those with inter-iteration dependences.

$$\begin{aligned}\mathrm{NPST}(\mathrm{node}) &= \max_i(\mathrm{NST}(\mathrm{pred}_i) + \mathrm{DS}(\mathrm{pred}_i) + 1)\\ \mathrm{NET}(\mathrm{node}) &= \mathrm{II}-1 + \min_i(\mathrm{NST}(\mathrm{pred}_i) + \mathrm{DS}(\mathrm{pred}_i) + 1)\end{aligned} \tag{5}$$

Based on slot freedom of each node on the formed ordered list, we schedule them in a 2-dimensional kernel starting from their NPST. When it is not feasible to schedule an operation due to a unit or write conflict, other possible units are tried or another cycle in [NPST + 1; NET] is used in a backtracking mode. The algorithm is lifetime-sensitive, integrating an accumulative measure of how much time the returned operands remain in core registers before being used. This criterion at a graph node is defined as the total lifetimes since results are being returned by all its predecessors before being consumed by this node; it is expressed in (6) (NRS ["node result size"] is the size in terms of core-registers for a node's results). Notice that this provided lifetime is weighted by the amount of registers that are presented and pended for use. The scheduling algorithm should minimize the overall accumulative lifetime which takes into consideration every node of the graph; in this case, register usage is in general also minimized.

$$\begin{aligned}\mathrm{lifetime}(\mathrm{node})\\ &= \sum_i \mathrm{NRS}(\mathrm{pred}_i)\Big(\mathrm{NST}(\mathrm{node}) - \mathrm{NST}(\mathrm{pred}_i)\\ &\quad - \mathrm{DS}(\mathrm{pred}_i)\Big)\end{aligned} \tag{6}$$

The total length of the accumulative lifetime divided by II gives a lower bound on register pressure, denoted AvgLive. Despite the fact that a minimized AvgLive gives smaller register usage, it does not necessarily provide the lowest. For example, a found schedule with a total lifetime of 110 requires a minimum of 17 registers, while another schedule with a lifetime of 118 required only a minimum of 16 registers. A better register lower bound is computed considering overlapped lifetimes over II cycles, getting an array (called LiveVector) of II elements as described in [33]. The maximum among the LiveVector values was named the MaxLive, which is a precise lower bound measure of the number of needed registers. It was shown in [33] that a schedule requires at most MaxLive + 1 registers.

Our algorithm merges the calculation of the MaxLive in the search process, informing of register pressure at any partial schedule. This serves us to efficiently cut off useless branches in the state space, reducing significantly the scheduling time. The pseudo-code of our scheduling algorithm is given in Fig. 15.

As the state space cannot be scanned entirely due to its NP-Complete nature despite MaxLive cut-offs, we propose to make smaller successive searches with different starting points on the search space. This is having the advantage to make finding a better schedule faster. Indeed, if we are not able to find a better solution within a specified amount of backtracking tries (100*M* nodes as an example), then it is more likely because first placed operations constrain the efficiency and therefore must be changed. The algorithm starts then with an initial ordering in the scheduling array, subsequently, if the backtracking amount limit is reached, the scheduling array is re-ordered according to specific rules and the search process starts again using a new initial state; this sequence is repeated until a solution fitting available registers is found.

The re-ordering part tries to guarantee different initial schedules and a fast convergence rate. The main used criteria while sorting is that when an operation *v* is unscheduled, next operations to be rescheduled must be those who maximize their effect on this operation *v*. In order to illustrate this criteria; let's take an example and assume that the scheduling array is arranged as follows: {mw1, mw2, mw3, m0, m1, m2, m3, a0, a1, a2, a3, a3i1, a3i2, v0, v1, v2, v3, f1, f2, f3, sf0, sf1, sf2, sf3, w1, w2, w3}.

Next, we assume that operations until "sf3" were placed successfully into the kernel and found a valid slot. If the operation "w1" cannot be placed into the schedule (either because it presents higher MaxLive pressure, or no free place could be found within its slots freedom range), then the algorithm will reschedule the previous operation in the scheduling array which is "sf3" and checking if this enhanced scheduling opportunities for "w1". It will not have an effect, because "sf3" is not sharing the same resource unit as "w1" nor among its direct predecessors. Hence, rescheduling "sf3", "sf2", "sf1", or "sf0" merely leads to useless states; only "mw1" (its direct predecessor) or operations using the same unit resource could have a chance to make a valid placement for "w1". In order to avoid scanning useless states first, a better order should have been done (making "mw1" close to "w1" in the scheduling array for example).

Accordingly, re-ordering the scheduling array will take those considerations:

If an operation is *j*-indexed into this array, all its predecessors must be indexed less than *j* {1}
Each operation must be close as possible to its direct predecessors {2}
Each operation must be close as possible to operations using the same unit resource {3}
Operations with larger input/output sizes are more critical to re-ordering considerations {4}

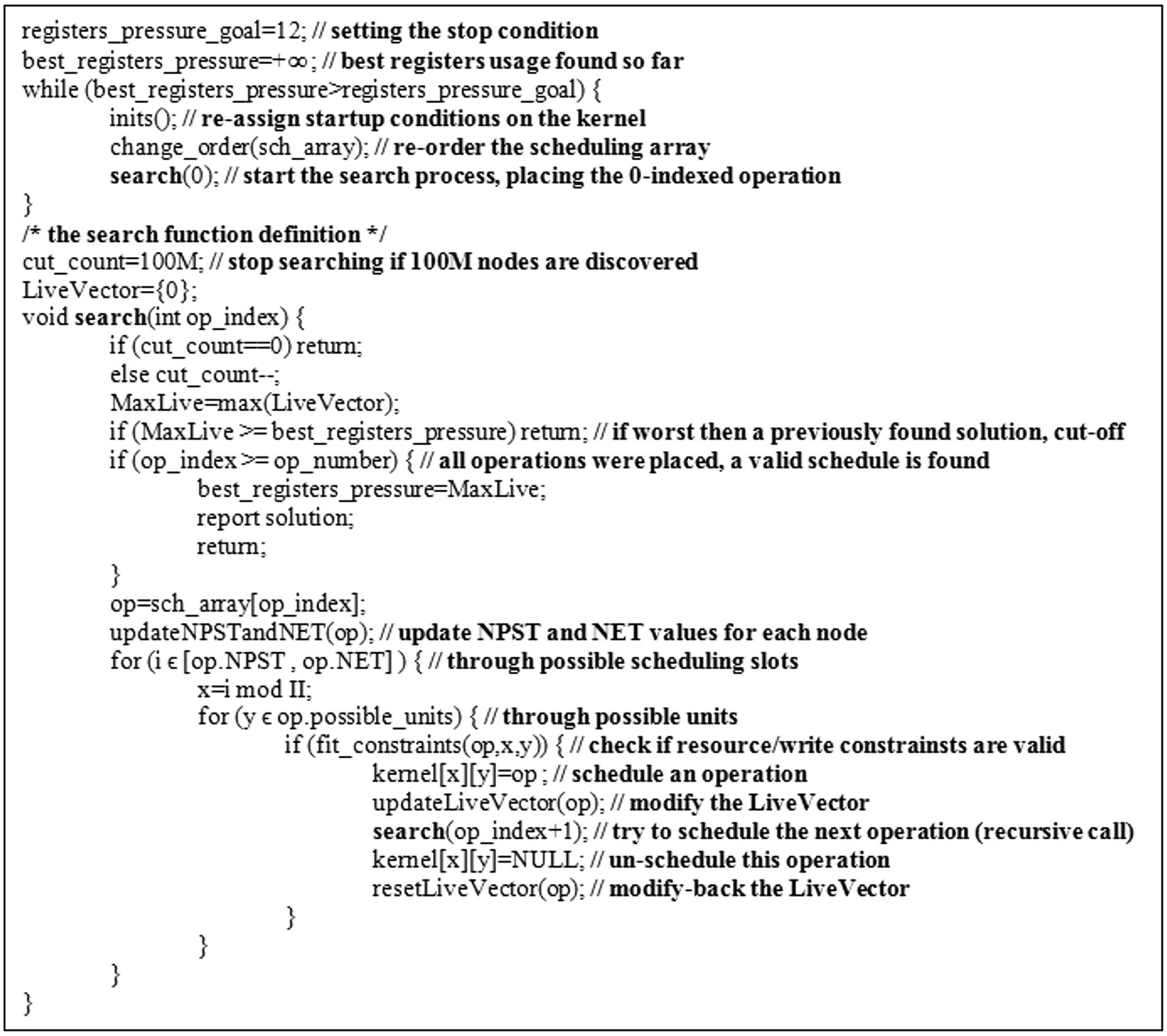
We next define for each graph node on the ordered list, a measure on its rescheduling easiness, denoted RF. It expresses how much a graph node meets the previously mentioned criteria regarding its indexing order into the scheduling array. Equation (7) defines the considered RF for a graph node operation op:

$$\mathrm{RF(op)} = \left(2\mathrm{op.id} - \frac{1}{\mathrm{op.pn}}\sum_{i=1}^{\mathrm{op.pn}} \mathrm{op.pred}[i].\mathrm{id} - \frac{1}{\mathrm{op.cn}}\sum_{i=1}^{\mathrm{op.cn}} \mathrm{op.conc}[i].\mathrm{id}\right)\mathrm{op.bs} \tag{7}$$

The ".id" field in (7) represents the ordering position of a node within the array. The *pn* and *cn* fields denotes respectively the number of predecessors of *op* and the number of concurrent operations using the same unit resource as *op* and which are indexed less than its index. The formula is scaled by the number of registers (buffers size ".bs") that are required by *op*.

The sum of all RF for every operation in the array reflects the ordering penalty (referred next by OrdP). A better ordering should have a minimized sum. The re-ordering routine generates a number of ordering possibilities; the one having the minimal penalty is picked and used next. This routine is as follows (assuming the array is presented with condition {1} verified):

```
function change_order(Sch_Array)
begin
loop
choose op: an operation from the Sch_Array;
min_i = index(op);
max_i = min(index(successors(op)));
//if no successor, set max_i to upper bound
choose a position s in the range [min_i, max_i-1];
move in array operation from position min_i to s;
compute the OrdP;
compare it to the best penalty found so far;
end loop
return the order having the lowest OrdP;
end
```



```
registers_pressure_goal=12; // setting the stop condition
best_registers_pressure=+∞; // best registers usage found so far
while (best_registers_pressure>registers_pressure_goal) {
        inits(); // re-assign startup conditions on the kernel
        change_order(sch_array); // re-order the scheduling array
        search(0); // start the search process, placing the 0-indexed operation
}
/* the search function definition */
cut_count=100M; // stop searching if 100M nodes are discovered
LiveVector={0};
void search(int op_index) {
        if (cut_count==0) return;
        else cut_count--;
        MaxLive=max(LiveVector);
        if (MaxLive >= best_registers_pressure) return; // if worst then a previously found solution, cut-off
        if (op_index >= op_number) { // all operations were placed, a valid schedule is found
                best_registers_pressure=MaxLive;
                report solution;
                return;
        }
        op=sch_array[op_index];
        updateNPSTandNET(op); // update NPST and NET values for each node
        for (i ϵ [op.NPST , op.NET] ) { // through possible scheduling slots
                x=i mod II;
                for (y ϵ op.possible_units) { // through possible units
                        if (fit_constraints(op,x,y)) { // check if resource/write constrainsts are valid
                                kernel[x][y]=op ; // schedule an operation
                                updateLiveVector(op); // modify the LiveVector
                                search(op_index+1); // try to schedule the next operation (recursive call)
                                kernel[x][y]=NULL; // un-schedule this operation
                                resetLiveVector(op); // modify-back the LiveVector
                        }
                }
        }
}
```

Fig. 15 Pseudo-code of our scheduling algorithm

The presented heuristic in this section was tested on a TI C66 VLIW core and generated an efficient schedule with II = MII and MaxLive = 12 during few minutes. Figure 16 shows the ordering penalty effect on the convergence speed. The resulting kernel form for the FFT/iFFT on C66 is shown in Fig. 11.

Our proposed scheduling algorithm aggressively minimized register usage, enhancing the MaxLive by a factor of 1.7 in the TI C66 core, toward the SMS scheduling method (which returned a MaxLive of 20), making an FFT completely without memory references of twiddle factors implementable.

The key parameters of our scheduling method are mainly the computed MII and U_m in Section 4.2.1, the number of symmetric clusters (denoted SymC) in the VLIW core (The method would schedule U_m/SymC butterflies per cluster, reducing the algorithmic problem size by a factor of SymC). The scheduling is also dependent on the VLIW instruction's delay slots, operand sizes, and their possible execution units and on the number of available core registers per cluster.

6 Implementation and experimental results

Subsequent implementation strategy for the FFT/iFFT was implemented in the high-end TMS320 C6678 VLIW DSP and in the 66AK2H12 DSP, using the Standard C66 Assembly. Data samples were single-precision complex floating point, with imaginary parts in odd indexes and real parts in even ones. During benchmarks, L1D/L1P was fully used as cache. Input/output buffers are stored into the memory L2 (512 Kbytes in C6678, 1 Mbytes in 66AK2H12). Moreover, the program code is mapped to

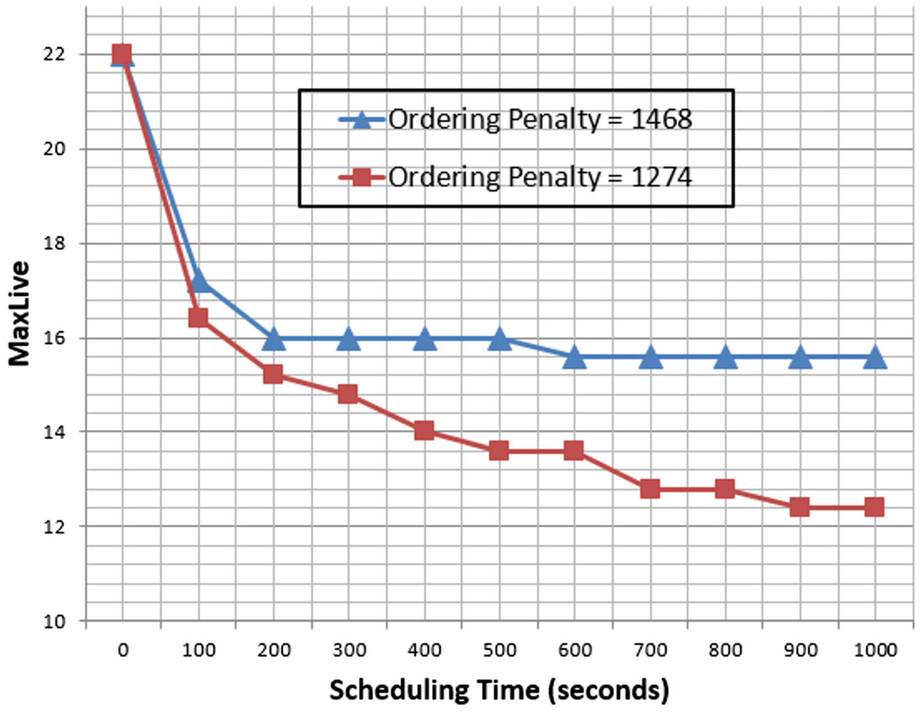


Fig. 16 The effect of the ordering penalty on the convergence speed

the local memory L2. Experimented FFT sizes are in the range [256, 64*k*], which correspond to most signal processing applications. The comparison is made with the most vendor-tuned linear assembly-optimized FFT of TI (found in the DSPLib version 3.1.1.1), with the TI compiler's optimizations all active (-o3, version 7.3.2).

A common efficiency measure for the FFT algorithm will be used, which is the number of cycles per pseudo-radix-2 butterfly (that we will denote as CPPR2B), defined by the FFT cycle count divided by $N\log_2(N)/2$. A perfect FFT implementation on the CorePac C66 would have a CPPR2B of 1; otherwise, it will be greater than 1. Performance comparison between our new FFT/iFFT (bit reversal included) and the optimized TI routines is presented in Fig. 17 and Table 2 (results are for 1 core of C6678, running at the frequency of 1 GHz).

Our presented FFT implementation shows great improvements over TI; the peak performance was reached for $N = 4k$, as the limited cache associativity made our special optimization to take place for $N = 8k$ and larger, inducing relatively less efficiency for integrating radix-2 stages. Small FFTs usually suffer from non-negligible overhead toward main processing. We are then able to reach an average gain of 50.56 % (2 times acceleration) over TI's routines, with a maximum performance of 1.119 CPPR2B (89.36 % of absolute efficiency). This obtained gain is explained by the proved 27 % gain in Section 4, the suppressed latencies of twiddle factors and by the other described optimizations.

Besides the speed performance, our FFT saves 50 % of input buffers toward conventional (TI) FFT routines (Table 3). Indeed, our scheme does not require twiddle factors to be stored nor to be referenced from external memory.

The bit reversal computation did not affect the performance for FFT sizes less than 8*k* (0 % increase due to our cache-efficient bit reversal optimization), while it increased the cycles count by an average of 18.4 % starting from 8*k* sizes; the loss represents the price of the time needed to process bit reversal separately for large FFTs. Besides, the proposed FFT/iFFT adaptation to the cache associativity is having at least a gain of 69 % in terms of efficiency. The inverse FFT was optimally reformed such that its performance overlapped with the FFT routine, despite the fact it included extra arithmetic operations.

Figure 18 shows the relative RMS error comparison between the new FFT and TI's one. The proposed FFT using 1 twiddle factor (tradeoff factor of 0) achieves for instance an RMS error of about 1E-05 (FFT size of 4*k*), against 5E-07 for TIs. Our FFT variant with a tradeoff factor of 1 (using log(*N*) pre-computed twiddle factors) achieved about a 10 times enhancement in terms of precision, preserving exactly the same presented speed performance. For most signal processing applications, this FFT accuracy is enough, and we can see for instance that the signal on FFT-noise ratio (SNR) we achieve in Fig. 18 for a 4*k*-FFT is about 107 dB, compared to

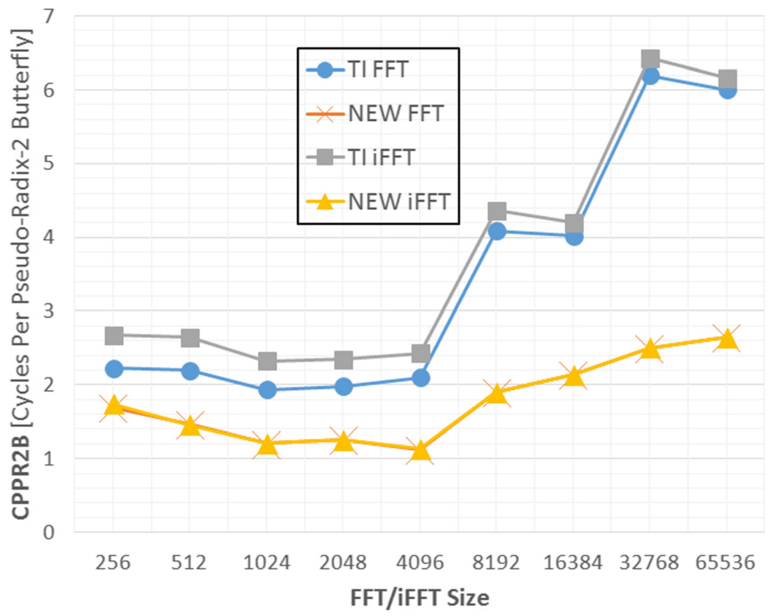


Fig. 17 Performance comparison between the new FFT and TIs in C66 cores

127 dB for the TI FFT and to 138 dB for the maximum achievable SNR of an IEEE-754 single-precision floating point. Worth to note that a 16-bit ADC converter has an SQNR of about 84 dB [40] (which is even less than the SNR due to FFT's computation accuracy); the SQNR is usually raised by coherent signal processing gains (e.g., a 4*k*-FFT processing can increase the SQNR by 33 dB or by 10 $\log_{10}(N/2)$ dB).

On the other hand, our scheduling heuristic generates the kernel codes of the FFT/iFFT and aggressively optimizes the number of cycles and registers' usage. It is able to compute the best schedule respecting tight pressure constraints with a fast convergence rate, overcoming results given by SMS by a factor of 1.7 (40 % gain) on the found MaxLive. The best generated schedule of instructions with a MaxLive = 12 was computed within 2- to 15-min range.

Furthermore, a 4*k*-sample floating-point FFT was performed in-chip during 2.6 μs within a 10-W power consumption in a TI 66AK2H12 DSP device, making a remarkable FFT implementation efficiency of 9.5 GFLOPS/watt. This makes it possible for use in several compute-intensive applications, such as radar processing [41]. Our work has been used within the official FFT library of Texas Instruments [42].

In contrast to previous works, on-the-fly generation of twiddle factors as in [43–45] used the CORDIC algorithm or related generation designs to compute the needed twiddle factors instead of performing ROM accesses. These techniques target hardware FFT designs in FPGAs or ASICs and are not applicable for CPU or DSP platforms. Indeed, the idea of generating twiddle factors using multipliers (or equivalent operations) in software for CPU/DSP is usually avoided, as it requires in most cases more latency than FFT schemes with pre-computed twiddle

Table 2 Performance comparison between the new FFT/iFFT and TIs

Implementation	FFT size	CPPR2B	Cycles	Relative gain over TI (%)
New FFT	1*k*	1.206	6175	37.62
TI FFT	1*k*	1.933	9899	-
New FFT	4*k*	1.119	27502	46.65
TI FFT	4*k*	2.097	51547	-
New FFT	8*k*	1.896	100979	53.63
TI FFT	8*k*	4.090	217782	-
New iFFT	1*k*	1.211	6198	47.89
TI iFFT	1*k*	2.323	11894	-
New iFFT	4*k*	1.130	27779	53.50
TI iFFT	4*k*	2.431	59745	-
New iFFT	8*k*	1.896	100977	56.59
TI iFFT	8*k*	4.368	232613	-

Table 3 Twiddle factor (TF) storage/reference comparison

Implementation	FFT size	Number of stored TF	Number of memory references due to TF
New FFT	512	1	1
TI FFT	512	512	511
FFT of [21] in C64x	512	254	127
New FFT	1*k*	1	1
TI FFT	1*k*	1024	1023
FFT of [21] in C64x	1*k*	510	255

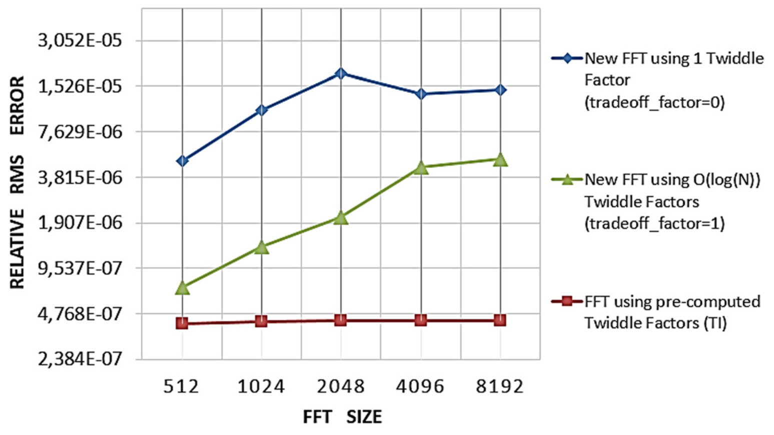


Fig. 18 RMS error comparison between the new FFT and TIs in C66 cores

factors. To the best of our knowledge, no published work proposed a software-efficient solution to do an FFT with in-generation of twiddle factors. The idea becomes possible with recent high-end VLIW processors, where we have to issue parallel instructions computing the twiddle factors in the masked time; however, it requires proper scheduling and low-level control on the execution pattern to be done successfully.

7 Conclusions

In the present paper, a new radix-2^2-based FFT/iFFT scheme is proposed to fit VLIW processors. This structure made a balance between the VLIW computation capabilities and the data bandwidth pressure, optimally exploiting parallelism opportunities and reducing memory references to twiddle factors, leading to an average gain of 51 % on efficiency toward the most assembly-optimized and vendor-tuned FFT on a high-end VLIW DSP. Our implementation methodology took into account the VLIW hardware resources and the cache structure, adapting the FFT algorithm to a broad range of embedded VLIW processors. On the other hand, a resource-constrained and register-sensitive modulo scheduling heuristic was designed to find the best low-level schedule to our FFT scheme, minimizing clock cycles and register usage using controlled backtracking, generating efficient assembly-optimized FFT with balanced resource usage.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

This research is sponsored in part by a Research and Development Contract from Thales Air Systems.

Author details

[1]LGECOS Lab, ENSA-Marrakech of the Cadi Ayyad University, Marrakech, Morocco. [2]Thales Air Systems, Paris, France.

Received: 23 October 2015 Accepted: 14 March 2016
Published online: 31 March 2016

References

1. JW Cooley, JW Tukey, An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**, 297–301 (1965)
2. GD Bergland, A radix-eight fast-Fourier transform subroutine for real-valued series. IEEE Trans. On Electroacoust. **17**(2), 138–144 (1969)
3. RC Singleton, An algorithm for computing the mixed radix fast Fourier transform. IEEE Trans. Audio Electroacoust. **1**(2), 93–103 (1969)
4. P Duhamel, H Hollmann, Split radix FFT algorithm. Electronics Letters **20**, 14–16 (1984)
5. D Takahashi, An extended split-radix FFT algorithm. IEEE Signal Processing Letters **8**(5), 145–147 (2001)
6. AR Varkonyi-Koczy, A recursive fast Fourier transform algorithm. IEEE Trans. on Circuits and Systems, II **42**, 614–616 (1995)
7. A Saidi, Decimation-in-time-frequency FFT algorithm. Proc. ICAPSS **3**, 453–456 (1994)
8. BM Baas, A low-power, high-performance, 1024-point FFT processor. IEEE J. Solid-State Circuits **34**(3), 380–387 (1999)
9. R Weber et al., Comparing hardware accelerators in scientific applications: a case study. IEEE Trans. Parallel Distrib. Syst. **22**(1), 58–68 (2011). doi:10.1109/TPDS.2010.125
10. T Fryza, J Svobodova, F Adamec, R Marsalek, J Prokopec, Overview of parallel platforms for common high performance computing. Radioengineering **21**(1), 436–444 (2012)
11. Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. Enabling an OpenCL compiler for embedded multicore DSP systems. In Proceedings of the 2012 41st International Conference on Parallel Processing Workshops (ICPPW '12). (IEEE Computer Society, Washington, DC, USA, 2012), p. 545-552
12. Francisco D. Igual, Guillermo Botella, Carlos García, Manuel Prieto, Francisco Tirado, Robust motion estimation on a low-power multi-core DSP. EURASIP Journal on Advances in Signal Processing. **99**, 1-15 (2013)
13. T. Fryza and R. Mego, Low level source code optimizing for single/multi/core digital signal processors, Radioelektronika (RADIOELEKTRONIKA), 2013 23rd International Conference, Pardubice, 2013, pp. 288-291. doi:10.1109/RadioElek.2013.6530933
14. JA Fisher, P Faraboschi, C Young, Embedded computing: A VLIW approach to architecture, compilers, and tools. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005), ISBN: 9780080477541.
15. V Zˇ Zivojnovi'c, Compilers for digital signal processors. DSP & Multimedia Technology Magazine **4**(5), 27–45 (1995)
16. J. P. Grossman, Compiler and architectural techniques for improving the effectiveness of VLIW compilation. [Online]. Available:http://www.ai.mit.edu/projects/aries/Documents/vliw.pdf [24-Mars-2016]
17. JA Fisher, Trace scheduling: a technique for global microcode compaction. IEEE Trans. Comput. **30**(7), 478–490 (1981)
18. L Monica, *Software pipelining: an effective scheduling technique for VLIW machines* (Proc. SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, 1988), pp. 318–328

19. B Ramakrishna Rau, *Iterative modulo scheduling: an algorithm for software pipelining loops*. Proc. 27th Annual International Symposium on Microarchitecture, 1994, pp. 63–74
20. M. Bahtat, S. Belkouch, P. Elleaume, P. Le Gall, Fast enumeration-based modulo scheduling heuristic for VLIW architectures, in 26th International Conference on Microelectronics (ICM), 2014, pp. 116-119, 2014. doi: 10.1109/ICM.2014.7071820
21. Y Wang, Y Tang, Y Jiang, JG Chung, SS Song, MS Lim, Novel memory reference reduction methods for FFT implementation on DSP processors. IEEE Trans. Signal Process **55**, 2338–2349 (2007). doi:10.1109/TSP.2007.892722
22. Y Jiang, T Zhou, Y Tang, Y Wang, Twiddle-factor-based FFT algorithm with reduced memory access, in *Proc. 16th Int. Symp. Parallel Distrib. Process* (IEEE Computer Soc, Washington, 2002), p. 70
23. K.J. Bowers, D.E. Shaw Res, New York, NY, USA; R.A. Lippert, R.O. Dror, D.E. Shaw, Improved twiddle access for fast Fourier transforms. IEEE Trans. Signal Process. **58**(3), 1122–1130 (2010)
24. VI Kelefouras, G Athanasiou, N Alachiotis, HE Michail, A Kritikakou, CE Goutis, A methodology for speeding up fast Fourier transform focusing on memory architecture utilization. IEEE Trans. Signal Process **59**(12), 6217–6226 (2011)
25. M Frigo, SG Johnson, The fastest Fourier transform in the west, in *Proc. Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, 1998
26. M Frigo, A fast Fourier transform compiler. SIGLAN Not. **39**, 642–655 (2004)
27. S Johnson, M Frigo, A modified split-radix FFT with fewer arithmetic operations. IEEE Trans. Signal Process **55**(1), 111–119 (2006)
28. D Mirkovic, L Johnsson, Automatic performance tuning in the UHFFT library, in *Computational Science—ICCS 2001* (Springer, New York, 2001), pp. 71–80
29. AM Blake, IH Witten, MJ Cree, The fastest Fourier transform in the south. IEEE Trans. Signal Process **61**(19), 4707–4716 (2013)
30. M. Bahtat, S. Belkouch, P. Elleaume, P. Le Gall, Efficient implementation of a complete multi-beam radar coherent-processing on a telecom SoC, in 2014 International Radar Conference (Radar), pp. 1-6, 2014. doi: 10.1109/RADAR.2014.7060412
31. S He, M Torkelson, A new approach to pipeline FFT processor, in *Proc. IEEE Parallel Processing Symp*, 1996, pp. 766–770
32. J.M. Codina, J. Llosa, A. González, A comparative study of modulo scheduling techniques, Proceedings of the 16th international conference on Supercomputing ICS 02(2002), **13**(1), 97. ACM Press
33. RA Huff, Lifetime-sensitive modulo scheduling, In Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation. 258-267(1993)
34. AK Dani, VJ Ramanan, R Govindarajan, Register-sensitive software pipelining. Parallel Processing Symposium, 1998. (IPPS/SPDP, Orlando, FL, 1998), p. 194-198
35. J. Llosa, A. González, E. Ayguadé, M. Valero, Swing modulo scheduling: a lifetime-sensitive approach, PACT '96 Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques. (Boston, MA, 1996), p. 80-86
36. J Llosa, E Ayguade, A Gonzalez, M Valero, J Eckhardt, Lifetime-sensitive modulo scheduling in a production environment. IEEE Trans. Comput. **50**(3), 234–249 (2002)
37. J Zalamea, J Llosa, E Ayguade, M Valero, Register constrained modulo scheduling. IEEE Trans. Parallel Distrib. Syst. **15**(5), 417–430 (2004)
38. TMS320C6678, Multicore fixed and floating-point digital signal processor, Data Manual, Texas Instruments. SPRS691E. March 2014. [Online]. Available: www.ti.com/lit/gpn/tms320c6678 [25-Mars-2016]
39. M Tasche, H Zeuner, Improved roundoff error analysis for precomputed twiddle factors. J. Comput. Anal. Appl. **4**(1), 1–18 (2012)
40. JJ Alter, JO Coleman, Radar digital signal processing, Chapter 25 in Merrill I. Skolnik, Radar Handbook, Third Edition, (McGraw-Hill, 2008)
41. A Klilou, S Belkouch, P Elleaume, P Le Gall, F Bourzeix, MM Hassani, Real-time parallel implementation of pulse-Doppler radar signal processing chain on a massively parallel machine based on multi-core DSP and serial RapidIO interconnect. EURASIP Journal on Advances in Signal Processing **2014**, 161 (2014)
42. Texas Instruments. FFT library for C66X floating point devices, C66X FFTLIB, version 2.0. [Online]. Available: http://www.ti.com/tool/FFTLIB [25-Mars-2016]
43. Sang Yoon Park; Nam Ik Cho; Sang Uk Lee; Kichul Kim; Jisung Oh, Design of 2K/4K/8K-point FFT processor based on CORDIC algorithm in OFDM receiver, Communications, Computers and signal Processing, 2001. PACRIM. 2001 IEEE Pacific Rim Conference on, vol. 2, no., pp. 457,460 vol. 2, 2001. doi:10.1109/PACRIM.2001.953668
44. T Pitkänen, T Partanen, J Takala, Low-power twiddle factor unit for FFT computation. Embedded Computer Systems: Architectures, Modeling, and Simulation Lecture Notes in Computer Science **4599**(2007), 65–74 (2007)
45. JC Chi, SG Chen, *An efficient FFT twiddle factor generator* (Proc. European Signal Process. Conf, Vienna, 2004)