Architectural Synthesis of Multi-SIMD Dataflow Accelerators for FPGA

Yun Wu, *Member, IEEE* and John McAllister, *Senior Member, IEEE*

Abstract—Field Programmable Gate Array (FPGA) boast abundant resources with which to realise high-performance accelerators for computationally demanding operations. Highly efficient accelerators may be automatically derived from Signal Flow Graph (SFG) models by using architectural synthesis techniques, but in practical design scenarios, these currently operate under two important limitations - they cannot efficiently harness the programmable datapath components which make up an increasing proportion of the computational capacity of modern FPGA and they are unable to automatically derive accelerators to meet a prescribed throughput or latency requirement. This paper addresses these limitations. SFG synthesis is enabled which derives software-programmable multicore single-instruction, multiple-data (SIMD) accelerators which, via combined offline characterisation of multicore performance and compile-time program analysis, meet prescribed throughput requirements. The effectiveness of these techniques is demonstrated on tree-search and linear algebraic accelerators for 802.11n WiFi transceivers, an application for which satisfying real-time performance requirements has, to this point, proven challenging for even manually-derived architectures.

Index Terms—Field programmable gate array (FPGA), dataflow, signal flow, architectural synthesis, single-instruction multiple-data (SIMD)

1 Introduction

Field Programmable Gate Array (FPGA) offer enormous computational capacity and distributed memory resources for high-performance, low-cost realisation of signal, image and data processing [1], high performance computing and big data analytics [2] and industrial control [3] operations. As custom computing devices, FPGA typically host *accelerators*—components whose circuit architecture is tuned to realise a specific function with performance and cost well beyond that available via software-programmable devices, such as multicore processors or graphics processing units.

To achieve these benefits, accelerators have traditionally been developed manually at Register Transfer Level (RTL) [4]. This low level of design abstraction enables highly efficient results, but imposes a heavy development load made increasingly unproductive as the scale of modern FPGA devices increase. *Architectural Synthesis* (AS) eases this burden by automating the derivation of pipelined accelerators from Signal Flow Graph (SFG) models and has proven highly successful in producing high-performance, efficient results [1], [5].

In the context of modern FPGA, current SFG-AS approaches have two shortcomings. First, the accelerators they produce are networks of fixed-function components, such as adders, multipliers or dividers. However, modern FPGA increasingly rely on multi-functional or programmable components, such as the DSP48E1 slice in Xilinx FPGA [6], to provide computational capacity. When used to realise fixed-function components, multiple of these are required to affect different operations where otherwise one would suffice, leading potentially to increased resource cost. No current SFG-AS approach can harness these components' programmability. Additionally, when designing for an industrial operating context or for standards-based systems, accelerators need to meet a prescribed throughput or latency. No current SFG-AS approach can automatically derive accelerators to meet such requirements and as a result, iterative cycles of time-consuming FPGA place-and-route to refine results to meet a required performance. This is a highly time-consuming process and a major barrier to high-productivity accelerator design.

This paper addresses these shortcomings. Specifically, three principal contributions are made:

1) A novel SFG-AS approach is presented which derives accelerators composed of custom multicore SIMD processor architectures utilising the programmable datapaths on modern FPGA.
2) It is shown how, via off-line estimation of multicore performance and compile-time application analysis, accelerators may be automatically produced which meet a pre-defined throughput requirement.
3) Automatic AS of accelerators with demanding real-time requirements is demonstrated by application to the design of transceivers for 802.11n WiFi.

The remainder of this paper is structured as follows. Sections 2 and 3 outline the multi-SIMD accelerator design problem, before Sections 4 and 5 describe the synthesis approach and Section 6 applies this to the design of 802.11n transceiver accelerators.

• *The authors undertook this work at the Institute of Electronics, Communications and Information Technology (ECIT), Queen's University Belfast, Belfast BT7 1NN, United Kingdom.*
E-mail: {yun.wu, jp.mcallister}@qub.ac.uk.

Manuscript received 1 Sept. 2016; revised 31 July 2017; accepted 21 Aug. 2017. Date of publication 29 Aug. 2017; date of current version 8 Dec. 2017.
(Corresponding author: John McAllister.)
Recommended for acceptance by T. El-Ghazawi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2746081

2 Background

Modern FPGA boast enormous on-chip computation, distributed memory and communications resources. For example, Xilinx's Virtex-7 FPGA family offer per-second access to up to 7×10^{12} multiply-accumulate (MAC) operations and 40×10^{12} bits/s of memory via programmable DSP48E1 [6], Look-Up Table (LUT) and Block RAM (BRAM) [7] resources. Along with the abundance of on-chip registers on modern FPGA, these resources are ideal for creating high-throughput, deeply-pipelined accelerators [1], [2], [8]. However, to date accelerators have been designed at RTL, a low-level, manual process made increasingly unproductive as the scale of modern FPGA increases.

Currently, two popular classes of approach address this productivity problem by adopting more abstract design entry points. High-Level Synthesis (HLS) translates programs written in popular software languages, such as C, C++ or OpenCL, to accelerators [9], [10], [11]. These tools derive RTL circuit architectures from the input and allow a designer to manipulate performance and cost by transforming the C source via, for example, loop unrolling, or by issuing synthesis directives. They support bit-true hardware data-types for arithmetic and automatically generate RTL code, frequently via the use of advanced scheduling and resource sharing in order to affect performance and cost.

An alternative approach uses AS techniques to derive accelerators from SFG models described in tools such as MATLAB Simulink. Exemplified by tools such as Xilinx's System Generator, Altera's DSP Builder and Synopsys' Synplify DSP, these empower a designer to specify the behaviour of an RTL component before generating code in the form of VHDL or Verilog. They support bit-true hardware datatypes, automatic RTL code generation, close integration with vendor synthesis and place-and-route tools and hardware-in-the-loop emulation. The SFG models created are also ideal for application of AS transformations such as automatic pipelining/retiming and graph folding or unfolding to trade the performance and cost of the accelerator [1], [5].

Regardless of the approach chosen, however, some restrictions are apparent. Consider the designer's concern: to realise a given function, on a given FPGA device, with a throughput or latency which is prescribed either by an industrial operating context, or by standards to which the equipment of which it is a part must comply. In this scenario, SFG-AS and HLS tools' capabilities are currently lacking in two important ways. Whilst transformation techniques such as retiming [12] and folding or unfolding [5], [13] allow accelerator performance to be traded with resource or energy cost [1], [13], [14], current approaches provide only partial support for deriving accelerators with a given performance. This is because they measure performance and cost in terms of abstract units such as clock cycles (latency), samples/cycle (throughput) or number of arithmetic components [13]. This puts the effect on actual performance and cost in doubt, since facets such as number of LUTs, or the length of each clock period cannot be accurately estimated until the entire, highly time-consuming FPGA synthesis toolchain, including place-and-route, has been traversed. Hence creating an accelerator of a given performance is a very unproductive, manual process.

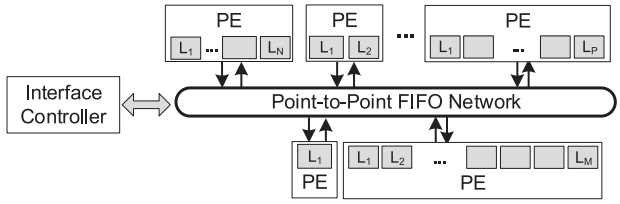


Fig. 1. multi-SIMD architecture template.

Furthermore, all of these techniques derive RTL circuits composed of fixed-function components, such as arithmetic components, buffers or switches. However a substantial proportion of the computational resource on modern FPGA is increasingly made up of components which are multi-functional or even programmable, such as the DSP48E1 on Xilinx Virtex-7 FPGA. If these are restricted to performing only one operation each, accelerators of greater cost may result than otherwise necessary. To the best of the authors' knowledge, no current AS approach addresses either of these shortcomings.

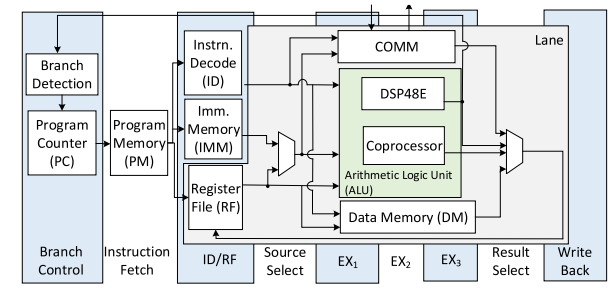
The programmable components which these processes should target require both control logic and memory resource to store and manage delivery of instructions and operands. These structures evoke the notion of software-programmable processors, the use of which on FPGA has been growing in recent times and has evolved into intermediate fabrics or overlays [15], [16], [17]. These take a wide variety of forms, including vector processors [18], [19], GPU-like structures [20], [21], [22] or domain-specific processors [16], [23]. These are all founded on components such as the DSP48E1 but impose large resource and performance overheads to enable program and data control. To enable efficient accelerators, these overheads must be minimised via a *soft* design approach which customises their structure to the workload at hand. An approach which adheres to this philosophy is described in [24], [25]. Here, very fine-grained processors are used as building blocks for large-scale multi-SIMD structures whose architectures are tuned to the workload. This approach has been shown to support accelerators with performance and cost which are highly competitive with those from libraries such as Xilinx's Core Generator or Spiral [26]. However, there is no technology to automate their generation.

This paper devises an SFG-AS approach which derives custom multi-SIMD processors built around programmable on-chip computation units. Section 3 introduces the target architectures in more detail.

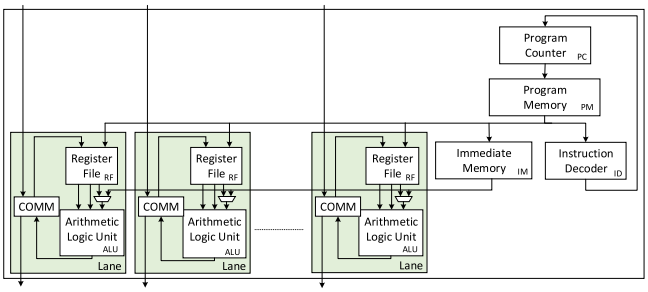
3 Heterogeneous Multi-SIMD for FPGA

3.1 FPGA Processing Elements

In [24], [25] is described an approach to the realisation of FPGA accelerators for signal, image and data processing using a template architecture shown in Fig. 1. Accelerators are realised using networks of Processing Elements (PEs)—software-programmable SIMD soft processors. The execution of each SIMD is decoupled from all others and communication is via point-to-point links. The structure of the network, the communications links and the widths of each PE are customisable at design time to maximise performance and minimise cost for the workload at hand.



(a) FPE - SISD Mode



(b) FPE - SIMD Mode

Fig. 2. The FPGA processing element.

To enable highly efficient accelerators, two key features are demanded of the PEs. They must be *lean*, incurring very low resource cost, to enable scalability to many hundreds of units for complex accelerators; associated with this requirement is the need for *standalone* operation—the ability to process data, access and manage memory and communicate externally without the need for a host processor. The FPGA PE (FPE) [24] is a RISC load-store PE which fulfils these requirements; SIMD and SISD (i.e., single-lane SIMD) variants of the FPE are shown in Fig. 2. The FPE includes only vital components—a program counter, program memory, instruction decoder, register file, branch detection, data memory, immediate memory and an arithmetic logic unit based on the DSP48E1 in Xilinx FPGA [6]. A COMM module allows direct insertion/extraction of data into and out of the FPE pipeline. In addition, the FPE's architecture is highly configurable for tuning to a specific workload [24].

By ensuring absolute lowest cost, economies of scale enable significant multicore resource cost savings. The price for this efficiency, however, is flexibility—the FPE is not a run-time general-purpose component because its architecture is highly tuned to the application at hand. In addition, it is domain-specific, enabling very high performance for certain types of operations, with performance degradation for others [25]. The benefit, however, is very high performance; a 16-bit SISD FPE on Xilinx Virtex 5 VLX110T supports 480 MMACs/s requiring 90 LUTs; this is just 14 percent of the cost of a general-purpose Xilinx Microblaze processor and 35 percent of that of the iDEA processor [23] on the same device. This efficiency enables processor-based accelerators for a range of applications whose performance and cost is highly competitive with hand-crafted accelerators. The structures which results are heterogeneous, as illustrated for an example symbol detector for 4×4 16-QAM Multiple-Input, Multiple-Output

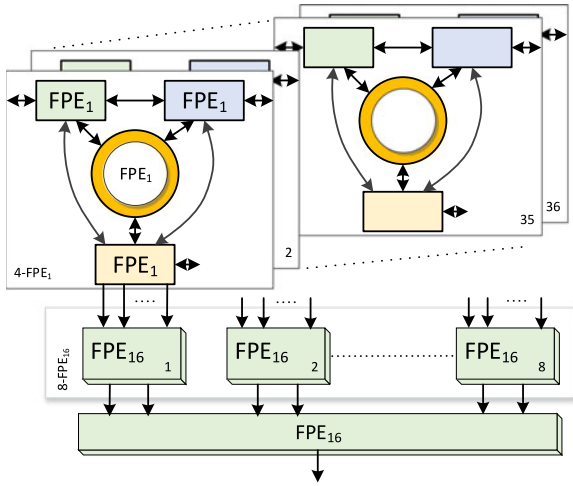


Fig. 3. FPE-based SD for 4×4 802.11n.

(MIMO) 802.11n transceivers in Fig. 3. This architectures includes clusters of MIMD structures (4-FPE_1) and nine 16-lane SIMD structures (FPE_{16}). This accounts for a total of 288 processing lanes, communicating via point-to-point data queues to realise the functionality required. The performance and cost of this architecture has been shown to be highly competitive with hand-crafted realisations of the same behaviour for this and a range of other functions [24], [25].

3.2 Synthesis of FPE-Based Accelerators

The goal, in this paper, is to generate a multi-FPE accelerator architecture from an application such that a prescribed throughput, expressed as a number of iterations n of the application per second, is satisfied. Accelerators based on the FPE promise high performance and efficiency, but a series of substantial design challenges must be overcome to automate their derivation to meet a given real-time performance. These are summarised in Fig. 4. A series of key subtasks are involved [4]:

- *Allocation* of a set of SIMDs to realise the application,
- *Partitioning & Binding* of application tasks to SIMDs and insertion of point-to-point communication links,
- *Scheduling* of the operations on each SIMD,
- *Estimation* of the performance of the result in order to ensure requirements are satisfied,
- *Code Generation* of source for each PE.

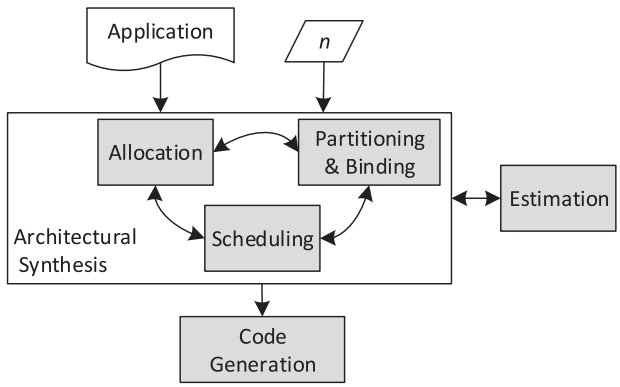
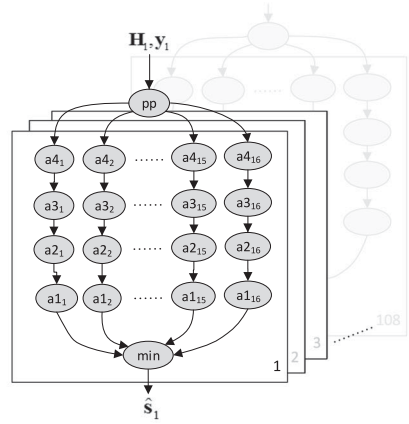
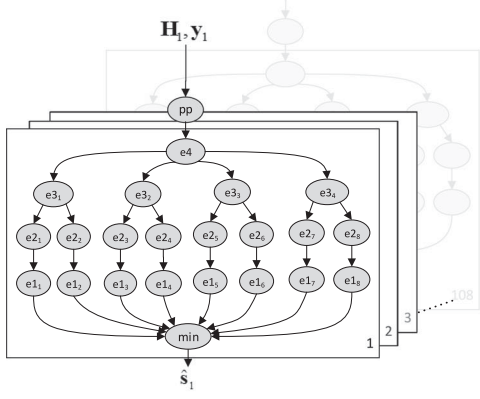


Fig. 4. FPE-based accelerator synthesis.



(a) FSD



(b) SSFE-$[1, 1, 2, 4]$

Fig. 5. FSD and SSFE SDF models.

Partitioning, scheduling and estimating the performance of application workloads for programmable multicore and GPU devices is an active topic of research [27], [28]. However, these works differ from that described here. For instance, in many cases they can reduce the scheduling load on the compiler by employing hardware scheduling circuitry [27], [28], [29]; this is not available in the FPE. In all cases, they do not have to allocate a processing resource, as is required for the FPE and their performance estimation, e.g., [30], does not extend to estimating the physical length of a clock cycle, as is required for FPGA.

The SFG modelling entry point is well-suited to synthesis problems such as this and has already been adopted in numerous FPGA AS tools as detailed in Section 2. It is a highly restricted form of dataflow, a domain of modelling languages which have been shown well-suited to rapid synthesis of digital signal processing operations for both multicore and FPGA [5], [31], [32], [33]. Specifically, SFG is a sub-class of synchronous dataflow [34], a decidable dataflow dialect [35] which has consistently demonstrated outstanding support for compile-time analysis and generation of efficient code. Its popularity has led to a considerable body of work in the areas of partitioning and binding, scheduling and code generation of dataflow applications for multiprocessors [32], [36], [37]. Hence, this paper focuses on the key novel aspects of this work: allocation of multi-SIMD PE architectures and mapping and scheduling of SFG tasks across PEs to meet a given real-time performance.

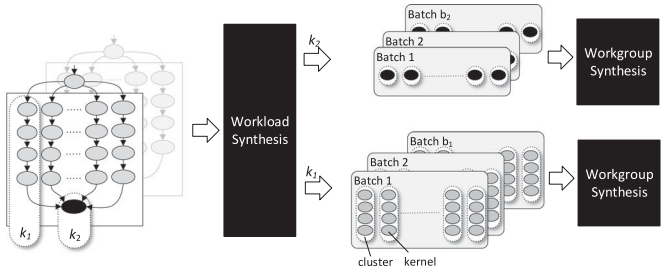


Fig. 6. Synthesis process overview.

4 Architectural Synthesis of Heterogeneous Multi-SIMD Accelerators

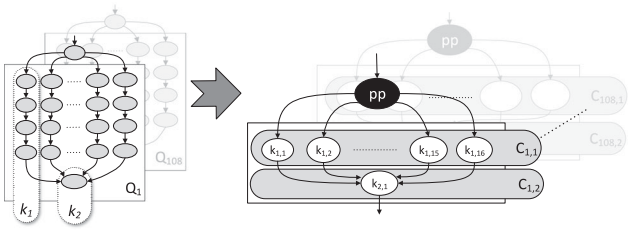
4.1 Synthesis Strategy

SFG models of two classes of tree-search Sphere Decoding (SD) operation—Fixed-Complexity Sphere Decoder (FSD) and Selective Spanning with Fast Enumeration (SSFE)-$[1, 1, 2, 4]$ - for 4×4 16-QAM MIMO for 802.11n Wi-Fi are shown in Fig. 5 and will be used to illustrate the synthesis process as it progresses. A SFG $G = (N, E)$ describes a set of nodes or actors N and a set of edges $E = N \times N$ - directed First-In, First-Out (FIFO) queues of data tokens. A node is said to *fire*, consuming/producing a pre-specified number of tokens, known as the *rate*, from each incoming/outgoing edge. In an SFG, all rates are 1 and are not quoted.

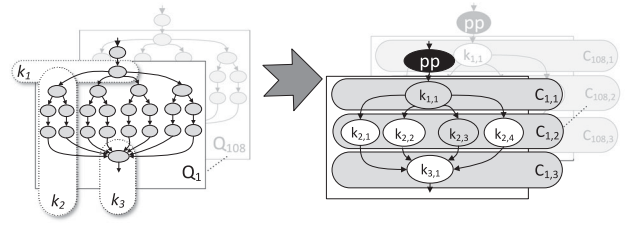
In Fig. 5, the SFG is composed of 108 subactors, each of which processes a single Orthogonal Frequency Division Multiplexing (OFDM) subdivision of the allocated frequency band. For each sub-band i, two pieces of data are input: a channel matrix $\mathbf{H}_i$ and a received symbol vector $\mathbf{y}_i$. In the cases considered in this paper, $\mathbf{H} \in \mathbb{C}^{4\times4}$ and $\mathbf{y} \in \mathbb{C}^{4\times1}$. Preprocessing is applied by *pp*, ordering the entries of both $\mathbf{y}$ and $\mathbf{H}$ according to the distortion on each path through the wireless channel, before an equalised version of $\mathbf{y}$ ($\mathbf{y}_{eq}$) is produced. The resulting data are then refined to an estimate $\hat{\mathbf{s}}$ of the transmitted symbol vector $\mathbf{s} \in \mathbb{C}^{4\times1}$ by a sequence of Euclidean distance cost functions $a1_i - a4_i$ in Fig. 5a ($e1_i - e4$ in Fig. 5b) Further details of these algorithms are available in [38], [39].

Each of the SFGs in Fig. 5 contain many instances of actors of a restricted range of classes, replicated in a very regular data/task parallel fashion. For instance, in FSD the sequence of actors $\{a4, a3, a2, a1\}$ forms all 16 branches of the SD tree, replicated 108 times to constitute 1,728 instances of this same sequence. Similarly, there are 16 data parallel instances of *min*, one per tree. These repeated parallel sequences are well suited to SIMD realisation, and we propose to exploit this feature to derive multi-SIMD realisations via a two-step process illustrated in Fig. 6.

As shown on the left of Fig. 6, the process commences with the SFG model and the definition of a set K of *kernels*. Kernels are considered the fundamental units of 'work', with the accelerator created to realise these kernels. This approach is in keeping with that of common heterogeneous computing languages such as OpenCL [40], where they are known as *work-items*, and CUDA. A kernel $k \in K$ is a subgraph of G such that G may be subdivided into a set of partitions $P = \{p_1, p_2, \ldots, p_n\}$, with each $p_i \in P$ an instance of a kernel $k \in K$ such that $p_1 \cup p_2 \cup \ldots p_n = G, p_i \cap p_j = \emptyset$ for all $i, j, i \neq j$, i.e., every actor in G is a member of precisely one



(a) Clustered FSD



(b) Clustered SSFE-$[1, 1, 2, 4]$

Fig. 7. Clustered FSD and SSFE results.

kernel instance. Kernels may take any form and are defined by the designer. However, in order to realise the most effective multi-SIMD realisations, these should be chosen to expose large numbers of similar, data parallel operations. In the FSD application, for example, both the FSD tree branches and the *min* actors, highlighted in Fig. 6 as k_1 and k_2 respectively, are ideal kernels.

From the SFG and kernel definitions are derived a *workload*. A workload is a sequence of kernel *batches*, with a multi-SIMD *workgroup* synthesised to process a prescribed number, n, of batches per second. Since all SIMDs in each workgroup execute the same kernel a distinct workgroup is required per kernel class; this paper illustrates the synthesis process for k_1 in Fig. 6.

4.2 Workload Synthesis

Consider the FSD model in Fig. 5a. Each OFDM sub-band contains multiple instances of the kernel k_1, all of which depend on data emanating from a *pp* node - $\mathbf{y}_{eq}$ and $\mathbf{H}$ - which are local to that sub-band. Hence, when realising the set of k_1 kernels, if two instances from the same sub-band are realised on different SIMDs or different lanes of the same SIMD, this local data will have to be stored in multiple different memories, increasing the total memory capacity required and the total FPGA resource cost. Conversely, if both are realised on the same SIMD lane, a substantial resource saving may result. This data locality is ensured by the SFG model's hierarchy and, as a result, it is important that this is maintained and exploited to guide the workgroup synthesis process to realise these kernels on the same SIMD resource. Accordingly, the workload is described as a batch of kernel *clusters*, formed to emphasise local communication and memory storage.

Realising this feature requires two capabilities. The graph G must be reformulated to express its behaviour in terms of the kernel set K, with any instance of a kernel $k \in K$ in G replaced by a single actor k, whilst kernels of the same class within the same composite node need to be clustered for batch formation. The effect of this clustering on the FSD and SSFE-$[1, 1, 2, 4]$ SFGs are shown in Fig. 7. Note that there are 108 disjoint FSD subgraphs, $Q_1 - Q_{108}$, each representing an OFDM subcarrier. Two kernels are identified: k_2 designates the *min* actor as a kernel, whilst k_1 identifies a branch of the FSD tree as a kernel. The SFG is factored to replace the subgraphs represented by each of these kernels with a single 'kernel' actor. In addition, the similar kernels in each disparate subgraph Q_i are composed into clusters and hence two clusters arise - C_1 and C_2, composed respectively of all instances of k_1 and k_2. Similarly, the three kernels identified in Fig. 7b result in three clusters for each Q_i. The SFG model reformulation process is performed as described in Algorithm 1.

Algorithm 1. DFG Clustering

```
1:  procedure ClusterDFG(G, K)
2:      i, j ← 1
3:      while ∃(Q ⊆ G) : (E(Q) ∩ E(G − Q) = ∅) do
4:          c_{i,j} ← ∅
5:          while j ≤ |K| do
6:              Q_k = Q ∩ k_j
7:              Q ← Replace(Q, Q_k, k_j)
8:              c_{i,j} ← c_{i,j} ∩ Q_k
9:              j ← j + 1
10:             i ← i + 1
        return G′, C
```

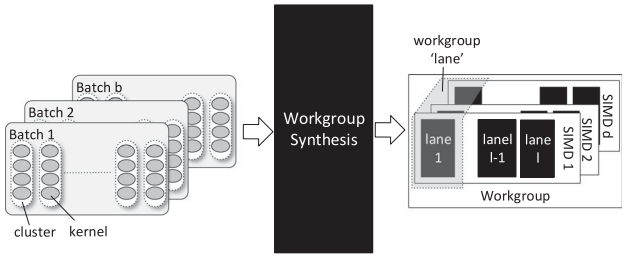
Input to this process are the set of kernels K and the SFG G. The goal is to derive G', a SFG of equivalent behaviour to G whose child actors are all kernels and members of K. In the process the set of kernel clusters C is also derived. The reformulation finds every instance of every kernel in G (line 6) by isolating its disjoint subgraphs $Q \subseteq G$ (line 3). All instances of each kernel k in Q (Q_k) are replaced with a single actor representing the kernel (line 7), with the set of kernels for each subgraph appended to the cluster definition (line 8). This process is repeated for every disjoint subgraph and every kernel type, with G' and C returned.

4.3 Design Space Scaffolding

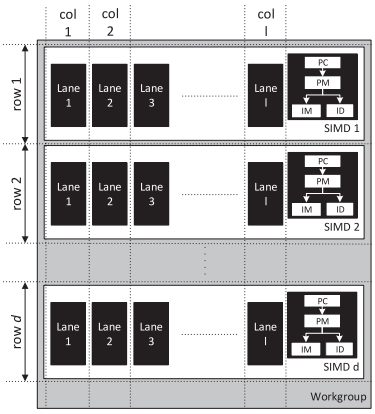
The workgroup synthesis strategy adopted is illustrated in Fig. 8a. A workgroup is synthesised for each class of kernel. It executes all instances of the kernel batches, where each batch is a set of clusters (as determined in Section 4.2) of sufficient size to meet the system throughput requirement. In deriving the workgroup, there are three key challenges: determining the batch size, the number of SIMDs and the width of each. To aid this process, a template workgroup structure is assumed, illustrated in Fig. 8b.

A workgroup is a two-dimensional SIMD structure, the rows of which are composed of SIMD units with column i ($i = 1, \ldots, l$) formed by the composite of lanes $(j, i), j = 1, \ldots, d$. To derive such a structure, d and l must be determined to execute a batch with a given throughput. The dimensions (d, l) can vary between:

- $(1, 1)$: one single-lane SIMD is employed to process w_i kernels sequentially,
- $(w_i, 1)$: multiple single-lane SIMDs are employed, with each SIMD realising a single kernel
- $(1, w_i)$: one multi-lane SIMD is employed, each lane of which realises a single kernel



(a) Synthesis Overview



(b) 2D multi-SIMD Resource Overview

Fig. 8. Workgroup structure & synthesis.

To guide the selection of the appropriate combination of rows/columns, two key observations may be made:

- To achieve highest efficiency, the kernel load of each column should be balanced so that no lane is idle awaiting others to finish. This implies that the number of kernels executed per column is an integer factor of the number contained in the batch.
- FPE performance scales linearly with number of lanes up to a width of 16, after which clock period constraints as a result of wide instruction broadcast imposes increasingly sublinear scaling [41].

A batch describes a subset of the clusters associated with each class of kernel; the set of viable batch sizes $S = \{s \in \mathbb{Z} : 1 \leq s \leq |C|\}$. For each viable batch size $s_i \in S$, a multi-phase *workload* may be defined as a sequence W, with each $w_i \in W$ determining the number of kernels executed during that phase of the sequence. Specifically, for each $s \in S$, $W = \{w_i\}_{i=1}^{\lceil \frac{|C|}{s_i} \rceil}$, where

$$w_i = \begin{cases} \sum_{j=s\cdot(i-1)+1}^{s\cdot i} |C_j| & \text{when} \quad i \leq \frac{|C|}{s} \\ \sum_{j=s\cdot(i-1)+1}^{|S|} |C_j| & \text{when} \quad i = \frac{|C|}{s} + 1. \end{cases} \tag{1}$$

Given these observations, a set L of candidate workgroup widths (i.e., number of columns) can be enumerated as the integer factors of the workload size, up to a limit of 16

$$L = \left\{ l \in \mathbb{Z}^+ : \left(\sum_{n=1}^{|C|} |c_i| \right) \% l = 0 \quad \text{and} \quad l \leq 16 \right\}. \tag{2}$$

Given this set of widths, the viable depths d can be determined by subdividing the batch across the workgroup columns and determining the number of SIMDs required by comparing estimates of the iteration rate to the requirement. This is achieved via Algorithm 2.

Algorithm 2. Workgroup Synthesis

```
1: procedure WorkgroupSynthesis(C, n)
2:     M, M_c ← ∅
3:     S ← {s ∈ ℤ : 1 ≤ s ≤ |C|}
4:     for each s ∈ S do
5:         W ← EnumerateWorkload(s)
6:         L ← EnumerateWidths(w_1)
7:         n' ← n × |W|
8:         for each l ∈ L do
9:             M' ← ∅
10:            M' ← Deploy(w_1, C, l, n')
11:            M_c ← M_c ∩ M'
12:    M ← SelectCandidate(M_c)
13:    return M
```

Two pieces of information allow the workgroup to be created: the kernel clusters C and n, which defines the number of iterations of the clusters required every second. The goal of workgroup synthesis is to determine a two-dimensional set M, with each element $m_{ij} \in M$ the sequence of kernels to be executed on the lane at row i, column j of the workgroup. To derive this, batch size and workgroup widths are successively enumerated and the batches deployed on the corresponding workgroups. The best of each of these options is chosen as the final deployment. Specifically, the set of all viable batch sizes S is enumerated (line 3 of Algorithm 2), and the corresponding sequence of batches W and workgroup widths L defined in lines 5 and 6, as in (1) and (2) (line 6) respectively. The workload is executed in $|W|$ phases and hence the iteration rate n is scaled accordingly (line 7). Then, for each candidate workgroup width, viable depths and workload mappings are derived (line 10, as described in Section 4.4) and appended to the set of candidate solutions M_c (line 11). The final result M is selected from this set (line 12).

4.4 Workgroup Derivation

The Deploy process maps a batch w of kernels onto a workgroup of a given width l such that a number of iterations per second of the batch n is achieved. The behaviour of this process is described in Algorithm 3.

Algorithm 3. Multi-SIMD Workload Deployment

```
1: procedure Deploy(w, C, l, n)
2:     A, M' ← ∅
3:     A ← ShapeWorkload(w, C, l)
4:     M' ← AllocateMapSchedule(A, n)
5:     return M'
```

There are two key steps: the batch is subdivided across the columns which make up the width of the workload (ShapeWorkload in line 3, described in Section 4.5), before the resulting arrangement, denoted by the set A is used to determine the number of SIMDs required in order to execute the kernels assigned to each column in satisfaction of n (line 4, described in Section 4.6). The cardinality of the resulting two-dimensional set M' defines the dimensions of

the SIMD array and whose entries define the sequence of kernels executed on each lane of each SIMD.

4.5 Workload Shaping

Workload shaping assigns kernels for execution on the columns of the two-dimensional workgroup; at the point of entry only the width of the workgroup (i.e., the number of lanes in each SIMD) is defined, with the number of SIMDs initially assumed to be one. The goal of this process is to subdivide the batch across a given number of columns (i.e., workgroup lanes), with the number of rows (i.e., SIMDs) to be later derived. In order to ensure that kernels sharing local data variables are assigned to the same SIMD lane, they must be assigned to the same workgroup column and hence the mapping of kernels to columns is guided by C according to Algorithm 4.

Algorithm 4. Workload Shaping

```
1: procedure ShapeWorkload(w, C, l)
2:   c_l ← w/l
3:   S ← C
4:   A ← ∅
5:   i, j, k ← 0
6:   while i ≤ |l| do
7:     R ← {q ⊆ s_j : |q| = min(|s_j|, c_l − |s_j|)}
8:     a_i ← a_i ∪ R
9:     s_j = s_j / R
10:    if |a_i| = c_l then
11:      k ← max(1, mod(k + 1, w))
12:    if |s_j| = 0 then
13:      j ← j + 1
14:    if k = 1 then
15:      i ← i + 1
16: return A
```

The ultimate aim is to derive a mapping of kernels from the batch w to workgroup lanes, deriving a set A each element $a_i \in A$ of which defines the kernels assigned to that lane. To derive this subdivision from w, the number of kernels per lane is calculated (line 2) and the kernels for each lane isolated (line 6). The assignment maintains column-local communication, i.e., kernels from the same cluster are assigned to the same column, as far as is possible. From each cluster are extracted kernels which number the lower of either the number of unmapped kernels in the cluster or the number required in order to fully load the current lane (line 7). These kernels are assigned to the current lane (line 8) and removed from the cluster (line 9). When a lane is fully loaded the next is considered (lines 10, 11); otherwise if all kernels in the cluster have been mapped the process repeats for the next cluster (line 12, 13).

4.6 Allocation, Mapping and Scheduling

Given the mapping of kernels to workgroup lanes, it remains to determine the number of SIMDs required in order to execute each lane's load to meet the system's throughput requirements. This is performed by a joint allocation/mapping/scheduling process which has three main objectives:

- Determine the number of SIMDs.
- Assign each kernel to a specific lane of a specific SIMD.
- Order the execution of kernels on each SIMD.

This requires a procedure with two inputs, a definition of the kernels assigned to each workgroup lane A, and a definition of the required number of iterations per second n times per second. The resulting two-dimensional set M' describes the sequence of kernels executed on each lane of each SIMD, derived via Algorithm 5.

Algorithm 5. Allocation, Mapping and Scheduling

```
1: procedure AllocateMapSchedule(A, n)
2:   d_max ← |a_1|, d_min ← 1
3:   M' ← ∅
4:   while 1 ≤ d_min ≤ d_max do
5:     d = d_min + ⌈(d_max − d_min)/2⌉
6:     k = |a_1| / d
7:     n_e = EstimateThroughput(k, l, d)
8:     if n_e > n then
9:       M' ← Map(A, d)
10:      d_max ← d − 1
11:    else
12:      d_min ← d + 1
13: return M'
```

There are, potentially, a significant number of options for the number of SIMDs—any integer number up to a maximum of $|a_i|$ - and a greedy design space pruning process is used to determine the appropriate value. Upper and lower bounds on the number of SIMDs, $d_{\max}$ and $d_{\min}$ are defined (line 2), with the range between these limits successively halved over multiple iterations. In each iteration, the performance of the mid-point of the range is estimated. In the case where it is too low, the upper half is chosen on the next iteration or, in case it exceeds the requirement, the lower half is chosen. In each iteration the number of kernels assigned to each workgroup row is determined (line 6) and its throughput estimated (line 7 - see Section 5). If the estimated throughput exceeds the requirement (line 8), the allocation is valid and the kernel load is mapped across the d SIMDs (line 9) before the upper bound on the search space is lowered to $d-1$ (line 10) and the process repeated to determine potentially lower-cost solutions. When performance is not sufficient, the lower bound $d_{\min}$ is increased to $d+1$ (line 11) and the process repeats until either the lower or upper bounds exceed their viable ranges. The result is the final workgroup derived which exceeds the performance requirement.

The process of mapping kernels to SIMD rows is a trivial subdivision of each $a \in A$ into d subsequences each of length $\lceil \frac{|a|}{d} \rceil$, where each subsequence describes the kernels to be executed on each row of the workgroup. This process is not described further here.

4.7 FSD Example

Fig. 9 illustrates the process of synthesising a workgroup to realise k_1 for FSD. As shown, there are 108 clusters, each containing 16 instances of k_1. Accordingly $S = \{1, 2, \ldots, 108\}$. For each $s_i \in S$, a workload can be derived. In the case where $s_i = 54$, a two-phase workload results, each phase of which executes 864 kernels. Given the processing of 54 clusters per batch, the viable widths of workgroup, i.e., the integer factors of the number of clusters, are given by $L = \{1, 2, 3, 6, 9\}$. For each $l_i \in L$ the number of workgroup rows may then be

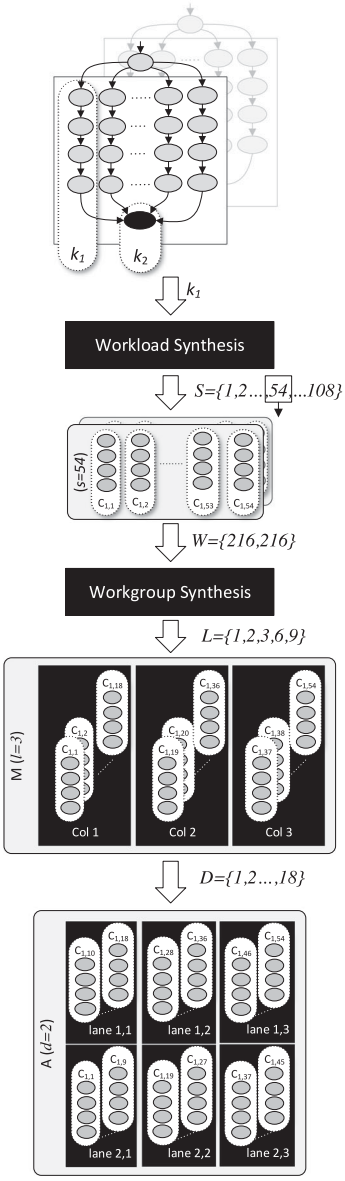


Fig. 9. Process overview for FSD.

defined as $D = \{d_i \in \mathbb{Z}^+ \cap [1, l_i]\}$. Fig. 9 illustrates the final arrangement when $d = 2$ and the kernel load for each workgroup column subdivided thereon.

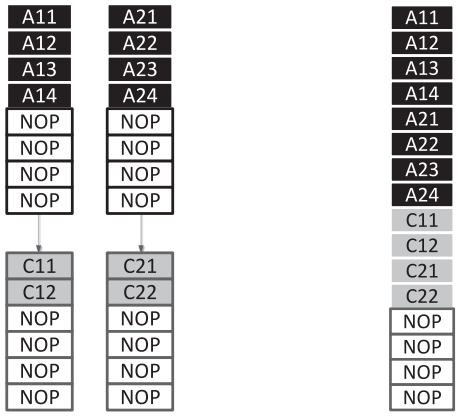
Key to this process is its ability to estimate the throughput of a given workload, on a given workgroup and to account for potential estimation inaccuracies. Techniques to facilitate both these objectives are described in Section 5.

5 Performance Estimation and Self-Correction

5.1 Throughput Estimation

To estimate throughput, two key metrics are required: the number of cycles required to execute the workload and length of each cycle, i.e., the clock period of the architecture.

Each SIMD executes a sequence of kernels and hence one prominent component of the throughput estimation problem is determining the number of instructions and cycles required to execute a given number of identical kernels. Given this information, the estimation process can employ



(a) Kernel Instruction Streams (b) Interleaved Kernels

Fig. 10. Kernel interleaving illustration.

any scheduling approach desired. Since the accelerator architecture exploits numerous copies of a single component (the FPE) in various SIMD configurations, the instruction stream for a kernel will be identical, regardless of on which component of the final architecture it is deployed. This allows pre-synthesis characterisation of the performance and cost of a kernel, a characterisation which may be used to enable the allocation process.

In order to reduce resource cost, forwarding hardware has been omitted from the FPE. In order, then, to avoid data hazards, NOPs must be inserted in the instruction stream realising a kernel in order to synchronise operand accesses. This leads to kernel instruction sequences such as that in Fig. 10a. Consider the resulting effect on the execution of a sequence of similar kernels by the FPE. Fig. 10 shows two example two-kernel workloads.

In both these cases, a series of Effective Instructions (EIs) is interspersed with NOPs for the purposes of data synchronisation. Assume that each kernel also requires r register file locations. In Fig. 10a, the two kernels may be executed sequentially, requiring only r RF locations; however, they may also be interleaved, with the EIs from one kernel occupying the NOPs from the other as in Fig. 10b. The interleaved version enables higher efficiency and throughput, but has increased RF cost. Hence each SIMD should interleave kernels as much as possible, so long as RF capacity constraints allow. The estimation problem is to determine the number of cycles required to execute a given multi-kernel workload, within a given constraint on r. This is determined by profiling the kernels, deriving instruction-level statistics of their computational operations and NOPs and combining these into a single cost metric.

Assuming an RF occupancy per kernel of r registers, then given a constraint on the number of RF locations r_c,[1] the maximum number of interleaved kernels f is given by

$$f = \lfloor \frac{r_c}{r} \rfloor. \quad (3)$$

The PM cost effect of interleaving successive kernels can be estimated by considering each kernel to be a sequence of

1. For the remainder of this paper, assume a maximum RF size of 64 locations

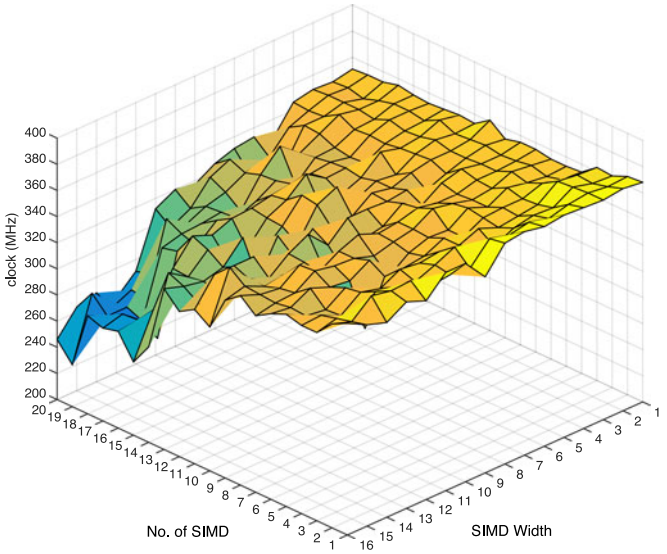


Fig. 11. Clock rate database for Virtex-5.

instructions subdivided into a sequence of blocks demarcated at $\{\text{NOP}, \text{EI}\}$ sequence boundaries - i.e., the first EI following a NOP represents the start of a new block. Each block consists of a set of EIs EI followed by a set of NOPs NOP and may then be represented by a coefficient γ

$$\gamma = \frac{|EI|}{|NOP|}. \tag{4}$$

Letting P_{IL} denote the maximum pipeline stage length,[2] kernel instruction statistics are categorised into P_{IL} catalogue sets depending on $(0 \leq \gamma \leq 1)$, $(1 \leq \gamma \leq 2)$, ..., $(\gamma = P_{IL} - 1)$. By defining two cost vectors, **a** and **b**, where a_i and b_i indicate respectively the number of EI and total instructions of a block in the ith catalog $(i \in [1, P_{IL}])$, then the PM size increment Δp of adding a further interleaved kernel is given by [42]

$$\Delta p = \sum_{i=1}^{k-2} a_i + k \cdot a_{k-1} - b_{k-1}. \tag{5}$$

Hence, for k kernels mapped to an FPE, the total PM cost is given by

$$\begin{aligned} p = &\lfloor \frac{k}{f} \rfloor [p + \Delta p(f-1) + \min(x, P_{IL} - 1)] \\ &+ (p + \Delta p[k \% f - 1]). \end{aligned} \tag{6}$$

The two additive terms in (6) respectively represent the total PM cost of the $\lfloor \frac{k}{f} \rfloor$ full interleaves and the final interleave, which may or may not be fully occupied. The value p denotes the number of cycles required to execute the multi-kernel workload for each FPE.

This analysis allows compile-time evaluation of the number of cycles required to execute a given set of kernels. The final performance in real-world terms depends not only on the number of cycles, but the length of each, as dictated by the clock period of the synthesised architecture. This period is determined by vendor place-and-route tools, such as Xilinx ISE or Vivado and the quality of the final result can be optimised by using additional intelligence to guide the process [43]. However, for this process the primary concern is the ability to estimate the final result, without undergoing the long delays associated with executing these functions. We need to be able to accurately estimate the length of each cycle that will result from any approaches such as these without actually executing them. This is achieved by pre-profiling, via RTL synthesis, varying numbers of SIMDs of varying width. Fig. 11 illustrates this profile for 1 to 20 SIMDs of each which has 1 to 16 lanes on Xilinx Virtex-5.

As this shows, the highest clock rate - approximately 370 MHz - is achieved by a single SISD processor with the lowest experienced for 20 SIMD processors with 16 FPEs. As shown in Fig. 11, the anticipated clock rates trends are observed—as the total resource realised on the device increases (represented by points towards the front left hand corner), clock rate reduces, as a natural result of the optimization algorithms executed by Xilinx ISE increasingly struggle to find low-cost/high-performance design space points as the scale of the gate-level netlist being mapped increases. Given this profiling and the estimation of the number of cycles required for workload execution, the throughput of a realisation, in iterations per second, may be estimated. Letting c_e denote the estimated clock frequency the estimated number of iterations n_e - used in Algorithm 5 to determine the viability of a realisation—is given by

$$n_e = \frac{c_e}{p}. \tag{7}$$

5.2 Self-Correction

At the point of estimation the number of instructions can, in fact, be measured rather than estimated. However, the clock frequency is a true estimate: the precise value cannot be known until after FPGA place-and-route is complete. At the proposed pre-synthesis point of estimation there is likely to be some error between the estimated and actual clock frequencies. Since this estimate is an intrinsic part of the design process the inherent inaccuracy may preclude the result from meeting the intended real-time performance.

Suppose that the estimated clock frequency is higher than the post-place-and-route actual frequency; this reduced clock frequency will lead to a reduced real-time performance, which in turn may be below the threshold performance target. In this case, allocation needs to be repeated to account for the discrepancy. To automatically derive a viable accelerator whilst accounting for the estimation discrepancy, the multi-phase synthesis process in Fig. 12 is employed.

As this shows, an iterative process adjusts the throughput target to account for inaccuracies in the estimated clock frequency c_e. If this exceeds the actual clock rate c_a and is sufficiently low that the actual number of iterations $n_a < n$, where n is the throughput requirement, then the threshold is adjusted (increased) to account for the differential, scaling by the ratio of the estimated and actual clock periods.

6 Experiments

To illustrate the capability of the proposed synthesis process, seven exemplar accelerators are addressed for 4×4 MIMO transceivers:

2. For the remainder of this paper, $P_{IL} = 6$

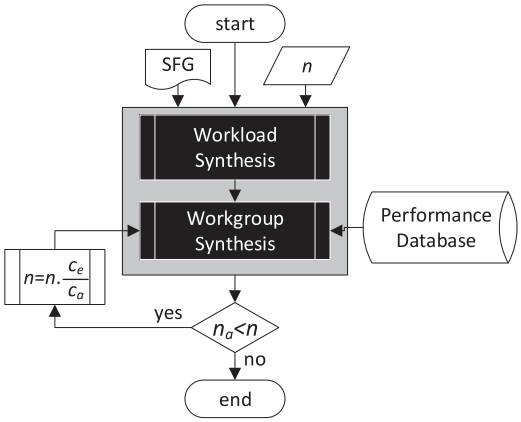


Fig. 12. Iterative synthesis process.

1) FSD, SSFE-$[1,1,1,4]$ and SSFE-$[1,1,2,4]$ tree-search SD
2) Zero-Forcing (ZF) and Minimum Mean Square Error (MMSE) equalisation
3) Sorted QR Decomposition (SQRD) pre-processing

We propose to evaluate the ability of the FPE AS approach by addressing the context of 802.11n, which demands 480 Mbps detection for FSD and SSFE and ZF/MMSE equalisation and 30×10^6 iterations/second SQRD - demanding requirements for even hand-crafted accelerators [24], [44]. This application has been chosen because it requires a range of operation types typical in signal, image and data processing—linear algebraic (matrix decomposition, matrix-vector and matrix-matrix multiplication) and tree-search operations, in a demanding real-time setting.

The SFG-AS process described in Sections 4 and 5 have been realised in a prototype, the behaviour of which is described in Fig. 13. In the main, XML is used for all input and intermediate data exchange, with the final result being

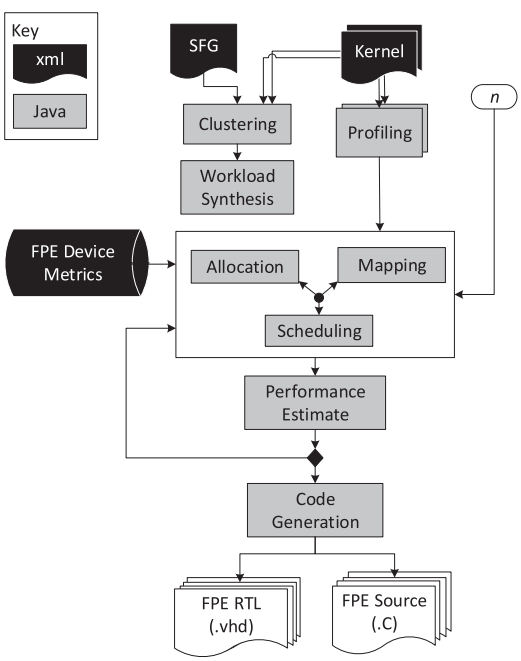
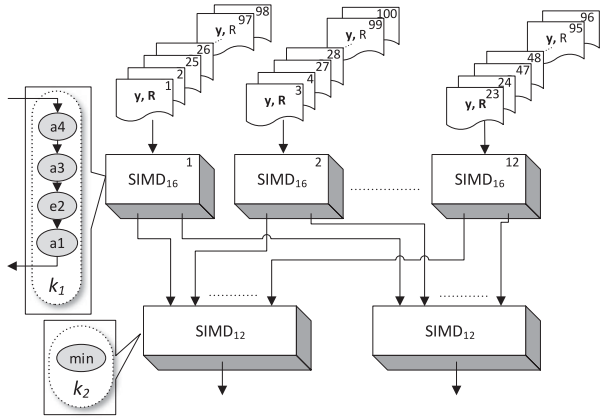
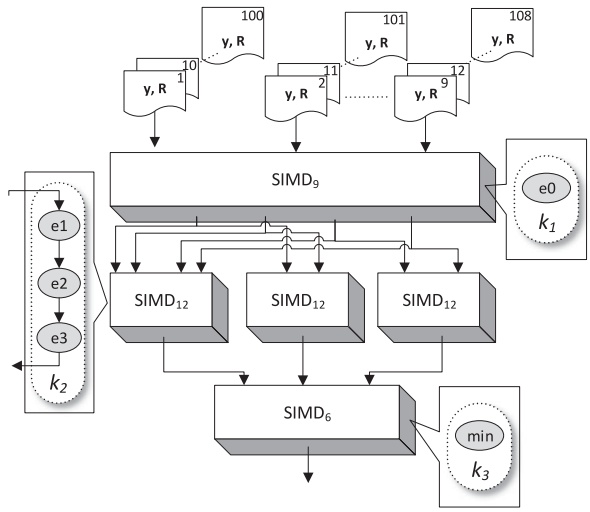


Fig. 13. Prototype SFG-AS tool structure.



(a) FSD multi-SIMD Architecture



(b) SSFE-$[1,1,1,4]$ multi-SIMD Architecture

Fig. 14. FSD and SSFE SIMD architectures.

VHDL and C sources describing the respective structure and executables for the multi-FPE accelerator derived. The intermediate processing stages match those in Sections 4 and 5 and are realised using Java. The C source for each FPE is compiled using a custom LLVM-based compiler, to produce assembly. The RTL source is translated to Xilinx Virtex-5 XC5VSX240T via ISE 14.2. In line with standard practise, to permit objective analysis of the performance and cost of the accelerators produced and comparison with existing and future approaches in the areas of HLS [10], [11], [13], [26], [43], FPGA-based processors [15], [16], [18], [19], [20], [22], [23], [24], [25], [43] and accelerators [1], [2], [13], [26], all performance and cost metrics are measured post-place and route, independent of a specific hardware platform.

6.1 Tree Search: FSD & SSFE

In order to realise FSD and SSFE-$[1,1,2,4]$ the SFG application models and corresponding kernels are respectively shown in Figs. 5 and 7. The kernels for SSFE-$[1,1,1,4]$ are illustrated in Fig. 14. Real-time operation for 4×4, 16-QAM 802.11n MIMO demands 480 Mbps throughput and is taken as the performance target. The resulting accelerators are itemised in Table 1. The FSD and SSFE-$[1,1,1,4]$ accelerators are depicted in Fig. 14.

TABLE 1
4×4 FSD Implementation Results

Modulation	16-QAM	64-QAM
SIMDs	14	25
DSP48E1	216	304
LUTs ($\times 10^3$)	31.1	154.42
Clock (MHz)	298	278
Throughput (Mbps)	483.2	506.4
Clock Est. (MHz)	289	276
Error (%)	3	0.72

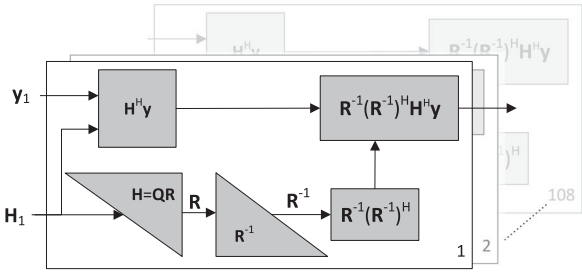


Fig. 15. ZF / MMSE SFG model.

The key features of the synthesis process are evident in the SIMD structures in Fig. 14. For FSD (Fig. 7a), two workgroups are created, one each for realisation of k_1 and k_2 in Fig. 6, with point-to-point FIFOs realising the dependencies between the two. For k_1 a workgroup of twelve 16-way SIMDs is realised, whilst for k_2, the workgroup consists of two 12-way SIMDs. Similarly, three workgroups are created for the three kernels which describe SSFE-$[1, 1, 1, 4]$ - a 9-way SIMD for k_1, three 12-way SIMDs for k_2 and a 6-way SIMD for k_3.

There are a number of notable aspects of the results in Tables 1 and 2. Immediately obvious is that the real-time performance requirements have been satisfied; to the best of the authors' knowledge, this is the first record of automatic derivation of multicore accelerators in satisfaction of a predefined performance requirement. In addition, it is worth noting the effectiveness of the proposed process in guiding the creation of each accelerator. In no case was the relative error in the estimated clock rate greater than 2 percent. This indicates that the pre-synthesis clock rate estimates were very accurate. Indeed, it is perhaps notable that in a number of instances, clock rate was underestimated.

6.2 Equalisation: ZF & MMSE

In MIMO communications, ZF or MMSE equalisation forms an estimate $\hat{\mathbf{x}}$ of the transmitted symbol vector $\mathbf{x}$ by forming the product of the received symbol vector $\mathbf{y}$ and an equalisation matrix $\mathbf{W}$

$$\hat{\mathbf{x}} = \mathbf{W} \cdot \mathbf{y}, \tag{8}$$

where $\mathbf{y} \in \mathbb{C}^{4\times 1}$ and $\mathbf{W} \in \mathbb{C}^{4\times 4}$ The equalisation matrix $\mathbf{W}$ takes different forms depending on whether a ZF or MMSE equalisation strategy is to be employed. For ZF

$$\mathbf{W}_{ZF} = \left(\mathbf{H}^H \cdot \mathbf{H}\right)^{-1} \cdot \mathbf{H}^H, \tag{9}$$

where $\mathbf{H} \in \mathbb{C}^{4\times 4}$ is the channel matrix and $\mathbf{H}^H$ denotes the hermitian transpose of $\mathbf{H}$. For MMSE equalisation,

$$\mathbf{W}_{MMSE} = \left(\mathbf{H}^H \cdot \mathbf{H} + \frac{\mathbf{I}_M}{\rho}\right)^{-1} \cdot \mathbf{H}^H, \tag{10}$$

where $\mathbf{I}_M$ is an identity matrix of order 4. The very high complexity of matrix inversion is the major challenge presented by this operation. To address this issue, QR decomposition is applied to $\mathbf{H}$ to produce

$$\begin{aligned} \mathbf{W} &= \left((\mathbf{Q} \cdot \mathbf{R})^H \cdot (\mathbf{Q} \cdot \mathbf{R})\right)^{-1} \cdot \mathbf{H}^H \\ &= \mathbf{R}^{-1} \cdot \left(\mathbf{R}^{-1}\right)^H \cdot \mathbf{H}^H, \end{aligned} \tag{11}$$

where both $\mathbf{Q}, \mathbf{R} \in \mathbb{C}^{4\times 4}$ According to this reformulation, the SDF application model for ZF and MMSE equalisation is shown in Fig. 15. As this shows, multiple operations are invoked, including QR decomposition of the channel matrix $\mathbf{H}$, followed by back-substitution to derive $\mathbf{R}^{-1}$ of the $\mathbf{R}$ matrix produced. Subsequently the products of $\mathbf{R}^{-1}$, its hermitian and $\mathbf{H}^H\mathbf{y}$ are formed to derive $\hat{\mathbf{y}}$. Real-time operation for 4×4 802.11n MIMO requires 480 Mbps throughput—the throughput and cost metrics obtained are described in Table 3.

It is again notable that, in both cases, real-time accelerators automatically result; to the best of the authors' knowledge, this is the first time this capability has been demonstrated for algebraic operations, such as the matrix triangularisation, inversion and multiplication operations. In addition, note again the effectiveness of the design process in estimating and refining the accelerator architecture. In the case of the MMSE accelerator, the estimation clock rate is only 6.7 percent in error. The situation is slightly deteriorated for ZF, where an 11.5 percent error in the initial estimate is encountered; whilst this is higher than any other estimate, it is still mild in absolute terms.

TABLE 2
4×4 16/64-QAM SSFE Implementations

	$[\mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{4}]$		$[\mathbf{1}, \mathbf{1}, \mathbf{2}, \mathbf{4}]$	
QAM	16	64	16	64
SIMDs	5	4	6	6
DSP48E1	45	33	80	49
LUTs ($\times 10^3$)	8.35	6.3	18.3	11.7
Clock (MHz)	355	357	354	353
Throughput (Mbps)	541.2	499.1	544.87	536.6
Clock Est. (MHz)	353	357	347	348
Error (%)	0.6	0	2.0	1.4

TABLE 3
4×4 ZF & MMSE Implementations

	ZF	MMSE
SIMD	13	24
DSP48E1	180	384
LUTs ($\times 10^3$)	31.6	73.1
Clock (MHz)	292	238
Throughput (Mbps)	507.69	591.6
Clock Est. (MHz)	330	255
Error (%)	11.5	6.7

TABLE 4
4×4 SQRD SD Preprocessing

	ZF	MMSE
SIMD	5	11
DSP48E1	72	180
LUTs ($\times 10^3$)	39.2	30.9
Clock (MHz)	347	314
Throughput (iterations/s)	31.1	30.4
Clock Est. (MHz)	345	335
Error (%)	0.6	6.3

6.3 Preprocessing: SQRD

In order to realise real-time ordering for 4×4 802.11n MIMO, the resulting accelerator has to operate at a rate of 30×10^6 iterations per second. SQRD merges QR decomposition of the channel matrix $\mathbf{H}$ with heuristic-based sorting to ensure that the decoding process addresses antennas in the correct order to account for the relative distortion experienced by each [45]. Table 4 reports both MMSE and ZF variants.

Once again, it is notable that these automatically derived accelerators meet the real-time performance requirement and that the estimation-based design process has been highly effective. For MMSE, the estimated clock rate and throughput are only 6.3 percent in error. Similarly, the ZF estimates are actually underestimates.

7 Conclusion & Future Work

This paper has presented an approach for AS of accelerators for modern FPGA which achieves two unique capabilities. By deriving custom multi-SIMD processors it can harness the programmable datapath resources which increasingly make up a substantial portion of the computational capacity of modern FPGA. Furthermore, it automates the generation of accelerators satisfying real-time performance requirements prescribed by their industrial operating context or as a result in standards-based equipment. This process is facilitated by offline characterisation of the performance of multi-SIMD topologies, compile-time evaluation of the cycle cost of SIMD programs and a self-correcting synthesis strategy which adapts to account for errors in the estimation process. When applied to the design of large-scale linear-algebraic (matrix triangularisation and multiplication) and tree-search operations, it automatically produces a series of accelerators capable of supporting real-time performance for 4×4 802.11n MIMO employing either 16-QAM or 64-QAM. This is a notable achievement since, on the same FPGA technology, implementations of the same operations has had to be enabled by hand-crafted RTL design, if indeed these previously existed - the authors are unaware of any work which enables real-time FSD employing 64-QAM, for instance. Furthermore, this paper targets Virtex 5 FPGA, but the techniques presented are applicable to later generations since the FPE 'virtualizes' the FPGA as it derives networks of FPEs and instructions for execution on each FPE and not the FPGA device architecture.

Despite the effectiveness of this approach, a series of further improvements could be made. For instance, it does not consider the resource cost of inter-processor communication and does not explore the potential for cost reduction via different mapping of the same application on an allocation. Similarly, automatically tuning the FPE RTL architecture to its functionality is not considered. Since the DSP48E slices targetted natively only support fixed-point arithmetic, the only way to support floating-point is via emulation, addition of floating-point co-processors next to, or in place of, the DSP48E in Fig. 2, or by combining this work with standard AS techniques which derive networks of fixed-function floating-point components. In addition, previous work [43] has shown the benefit of considering the nature of the processing architecture being realised when optimizing its mapping to the FPGA, and it is likely that a similar approach could yield increased performance and/or lower cost FPE-based accelerators. There is considerable performance/cost benefit to all of these considerations.

References

[1] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-Based Implementation of Signal Processing Systems*. Hoboken, NJ, USA: Wiley, 2008.

[2] W. Vanderbauwhede and K. Benkrid, *High-Performance Computing Using FPGAs*. Berlin, Germany: Springer, 2013.

[3] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in industrial control applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 2, pp. 224–243, May 2011.

[4] D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modelling, Synthesis and Verification*. Berlin, Germany: Springer, 2009.

[5] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Hoboken, NJ, USA: Wiley, 1999.

[6] *7 Series DSP48E1 Slice User Guide*, Xilinx Inc., San Jose, CA, USA, Aug. 2013.

[7] *7 Series FPGAs Memory Resources User Guide*, Xilinx Inc., San Jose, CA, USA, Jan. 2014.

[8] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Comput. Sci. Eng.*, vol. 14, no. 4, pp. 98–103, 2012.

[9] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2005.

[10] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[11] A. Canis, et al., "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[12] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991. [Online]. Available: http://dx.doi.org/10.1007/BF01759032

[13] Y. Yi and R. Woods, "Hierarchical synthesis of complex DSP functions using IRIS," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 5, pp. 806–820, May 2006.

[14] S. J. E. Wilton, S.-S. Ang, and W. Luk, "The impact of pipelining on energy per operation in field-programmable gate arrays," in *Proc. 14th Int. Conf. Field Programmable Logic Appl.*, 2004, pp. 791–728.

[15] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant FPGA compilation," *IEEE Embedded Syst. Lett.*, vol. 3, no. 3, pp. 81–84, Sep. 2011.

[16] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, Mar. 2016, pp. 1628–1633.

[17] C. Wang, J. Zhang, X. Li, A. Wang, and X. Zhou, "Hardware implementation on FPGA for task-level parallel dataflow execution engine," *IEEE Trans.| Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2303–2315, Aug. 2016.

[18] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux, "Vector processing as a soft processor accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, Jun. 2009, Art. no. 12.

[19] P. Yiannacouras, J. Steffan, and J. Rose, "Portable, flexible, and scalable soft vector processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 8, pp. 1429–1442, Aug. 2012.

[20] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *Proc. Int. Conf. Field-Programmable Technol.*, Dec. 2013, pp. 230–237.
[21] A. Al-Dujaili , F. Deragisch, A. Hagiescu, and W. F. Wong, "Guppy: A GPU-like soft-core processor," in *Proc. Int. Conf. Field-Programmable Technol.*, Dec. 2012, pp. 57–60.
[22] M. Al Kadi, B. Janssen, and M. Huebner, "FGPU: An SIMT-architecture for FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 254–263. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847273
[23] H. Y. Cheah, F. Brosser, S. Fahmy, and D. L. Maskell, "The iDEA DSP block based soft processor for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 1, Feb. 2014, Art. no. 19.
[24] X. Chu and J. McAllister, "Software-defined sphere decoding for FPGA-based MIMO detection," *IEEE Trans. Signal Process.*, vol. 60, no. 11, pp. 6017–6026, Nov. 2012.
[25] P. Wang and J. McAllister, "Streaming elements for FPGA signal and image processing accelerators," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 24, no. 6, pp. 2262–2274, Jun. 2016.
[26] P. Milder, F. Franchetti, J. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 2, pp. 15:1–15:33, Apr. 2012.
[27] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit," in *Proc. 43rd ACM/IEEE Int. Symp. Comput. Archit.*, Jun. 2016, pp. 609–621.
[28] M. Gebhart, et al., "A hierarchical thread scheduler and register file for energy-efficient throughput processors," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, pp. 8:1–8:38, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2166879.2166882
[29] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, Mar. 2011.
[30] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. 17th IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 382–393.
[31] E. Lee and T. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.
[32] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York, NY, USA: Marcel Dekker, 2000.
[33] S. Bhattacharyya, P. Murthy, and E. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *J. Very Large Scale Integr. Signal Process.*, vol. 21, no. 2, pp. 151–166, Jun. 1999.
[34] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[35] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of Signal Processing Systems*. Berlin, Germany: Springer, 2010.
[36] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.
[37] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Norwell, MA, USA: Kluwer, 1996.
[38] L. G. Barbero and J. S. Thompson, "Fixing the complexity of the sphere decoder for MIMO detection," *IEEE Trans. Wireless Commun.*, vol. 7, no. 6, pp. 2131–2142, Jun. 2008.
[39] Min Li, et al., "Selective spanning with fast enumeration: A near maximum-likelihood MIMO detector designed for parallel programmable baseband architectures," in *Proc. IEEE Int. Conf. Commun.*, May 2008, pp. 737–741.
[40] Khronos OpenCL Working Group, "The OpenCL C Specification," Sep. 2015.
[41] X. Chu, J. McAllister, and R. Woods, "A pipeline interleaved heterogeneous SIMD soft processor array architecture for MIMO-OFDM detection," in *Proc. 7th Int. Symp. Reconfigurable Comput.: Archit. Tools Appl.*, Mar. 2011, pp. 133–144.
[42] C. Zheng, J. McAllister, and Y. Wu, "A kernel interleaved scheduling method for streaming applications on soft-core vector processors," in *Proc. Int. Conf. Embedded Comput. Syst.*, Jul. 2011, pp. 278–285.
[43] C. E. LaForest and J. G. Steffan, "Maximizing speed and density of tiled FPGA overlays via partitioning," in *Proc. Int. Conf. Field-Programmable Technol.*, Dec. 2013, pp. 238–245.
[44] L. Ma, K. Dickson, J. McAllister, and J. McCanny, "QR decomposition-based matrix inversion for high performance embedded MIMO receivers," *IEEE Trans. Signal Process.*, vol. 59, no. 4, pp. 1858–1867, Apr. 2011.
[45] Y. Wu, J. McAllister, and P. Wang, "High performance real-time pre-processing for fixed-complexity sphere decoder," in *Proc. IEEE Global Conf. Signal Inf. Process.*, Dec. 2013, pp. 1250–1253.

Yun Wu received the BS degree in electronic and information engineering from Dalian Nationalities University, Dalian, China, in 2003, the MSc degree in circuits and system from Hunan University, Changsha, China, in 2007, the MSc degree in radio frequency communication systems from the University of Southampton, Southampton, U.K., in 2008, and the PhD degree in electronic engineering from Queen's University Belfast, Belfast, U.K., in 2014. Before that, he was a wireless algorithm engineer with ZTE Shanghai R & D Centre from 2008 to 2010. He is currently a research fellow with Queen's University Belfast. His current research interests include signal processing, processor architecture synthesis, and energy proportional computing. He is a member of the IEEE.

John McAllister (S'02-M'04-SM'12) received the PhD degree in electronic engineering from Queen's University Belfast, U.K., in 2004. He is currently a member of academic staff in the Institute of Electronics, Communications and Information Technology (ECIT), Queen's University Belfast, U.K. His research interests include custom computing for embedded streaming applications, including domain-specific languages, compilers, and computing architectures. He is a co-founder of Analytics Engines Ltd., a member of the Advisory Board to the IEEE Technical Committee on Design and Implementation of Signal Processing Systems (DISPS), a former associate editor of the *IEEE Transactions on Signal Processing*, chief editor of the IEEE Signal Processing Society Resource Center and a member of the editorial board of Springer's Journal of Signal Processing Systems. He is a senior member of the IEEE.