Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 961950, 20 pages
doi:10.1155/2012/961950

Research Article

High-Level Design Space and Flexibility Exploration for Adaptive, Energy-Efficient WCDMA Channel Estimation Architectures

Zoltán Endre Rákossy, Zheng Wang, and Anupam Chattopadhyay

Institute for Communication Technologies and Embedded Systems (ICE), RWTH Aachen University, Templergraben 55, 52056 Aachen, Germany

Correspondence should be addressed to Zoltán Endre Rákossy, rakossy@ice.rwth-aachen.de

Received 4 May 2012; Revised 15 October 2012; Accepted 24 October 2012

Academic Editor: René Cumplido

Due to the fast changing wireless communication standards coupled with strict performance constraints, the demand for flexible yet high-performance architectures is increasing. To tackle the flexibility requirement, software-defined radio (SDR) is emerging as an obvious solution, where the underlying hardware implementation is tuned via software layers to the varied standards depending on power-performance and quality requirements leading to adaptable, cognitive radio. In this paper, we conduct a case study for representatives of two complexity classes of WCDMA channel estimation algorithms and explore the effect of flexibility on energy efficiency using different implementation options. Furthermore, we propose new design guidelines for both highly specialized architectures and highly flexible architectures using high-level synthesis, to enable the required performance and flexibility to support multiple applications. Our experiments with various design points show that the resulting architectures meet the performance constraints of WCDMA and a wide range of options are offered for tuning such architectures depending on power/performance/area constraints of SDR.

1. Introduction

In a scenario of fast changing standards and process technologies, mobile devices increasingly rely on the software-defined radio (SDR) and cognitive radio [1, 2] concepts to achieve adaptability, flexibility, and spectral and energy efficiency. SDR implementation presents an interesting challenge for the architecture designers, namely, to develop an underlying hardware platform for SDR with fine balance of performance and flexibility. This demanding problem led to major research activity in recent years [3–12].

One of the key ingredients in the SDR architecture design is to determine the algorithmic kernels across various standards. While the kernel can be implemented in the most efficient manner, it can be retargeted according to different standards by means of tunable parameters or weak programmability. To that effect, the final architecture can be an ASIC, a reconfigurable platform or an application-specific processor. The complete system is often built by combining such accelerators, targeted for different blocks of a wireless standard [7]. Often it is of great importance how such a system can adapt to changes in the algorithms or standards, saving some of the engineering and development costs when standards change.

In this paper, we explore how flexibility can influence energy efficiency, area, and timing within the architectural design space of two wide-band code division multiple access (WCDMA) channel estimation algorithms. Channel estimation is an important basic block of advanced wireless standards, where hard deadlines must be ensured [13]. There are a wide range of proposed algorithms with significant performance/complexity differences which allow us to consider a scenario, where architectural flexibility can strongly influence the results: when less performance is acceptable during operation, switching to an inferior algorithm could save energy, since algorithm performance translates into complexity which directly influences energy consumption.

How much energy could be saved and what kind of flexibility is required in hardware for such adaptability is what we strive to answer in this paper. This triggers though an interesting design challenge as well, since design space exploration using traditional methods is often time-consuming and complex and is not desirable in today's short time-to-market.

Given the complexity of modern designs, various high level synthesis methodologies for quick architectural exploration are employed in industry and academia. These can be categorized into the following:

(i) *methodologies of direct translation* of high-level C language to hardware description like Calypto Catapult-C [14], GAUT [15] and Bluespec [16], yielding custom ASICs;

(ii) *customizable processor design* such as Tensilica [17] and ARC [18], using highly optimized blocks as components;

(iii) *architecture-description-language- (ADL-) based processor design*, creating fully flexible and custom processors, such as nML [19] and LISA [20, 21].

As one of our prime design goals is to explore flexibility, the above options fall short of our expectations. C-based HLS techniques offer no easy way to specify flexibility and custom processor designs often bring a lot of additional overhead in terms of fine-grained instruction execution. The inherent flexibility of ADL-based design flow would cover the required range, but great care must be employed when exploring its limitations.

As one of the prominent design flows, we adopt the commercially available and mature design flow of Synopsys Processor Designer [21] and propose additional modeling guidelines on how these tools can be exploited to tune the amount of flexibility in the design, from *just-enough* flexibility to employing *coarse-grained reconfigurability*. We present and evaluate these modifications at both flexibility grades by implementing and comparing the energy-saving scenario using two WCDMA channel estimation algorithms as target.

2. Organization of the Paper

The core content of this paper is structured as follows.

In Section 3 we describe and analyze the targeted application domain. Two algorithms for WCDMA channel estimation with differing complexity and performance are detailed and the complexity is exposed.

Section 4 mentions related work and existing implementation of WCDMA channel estimation in the literature.

Section 5 deals with our design methodology and explains the design flow. High level synthesis with LISA ADL is presented and improvements on the design flow are discussed.

Using the guidelines described, design space exploration at the two flexibility levels is performed in Section 6, yielding two classes of architectures.

Experimental results and comparison within architectural variants and across architectural classes are shown in Section 7, while discussing advantages and disadvantages of each.

Finally, Section 8 concludes the paper.

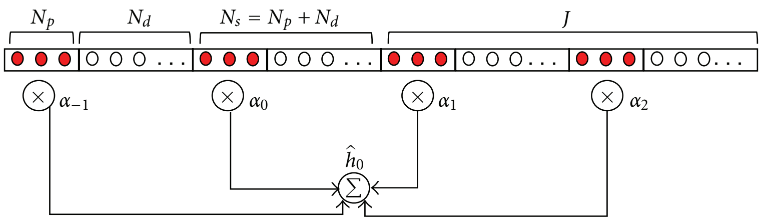
3. Target Applications

A critical part of wireless communication is maintaining a good level of signal-to-noise ratio (SNR) on the link. This is influenced negatively by multipath fading, mobile terminal speed relative to the base transmitter, scattering, shadowing, and so forth. To counter this, channel estimation (CE) is performed so that corrections can be done by considering dynamically altering channel conditions. CE constitutes an important building block for SDR, as this is used across multiple wireless standards.

There are 3 large classes in which one can categorize CE algorithms: (1) low-complexity, low-performance algorithms; (2) high-complexity, good-performance algorithms; and (3) extremely complex, iterative algorithms with near-optimal performance. While (1) deals with simple (linear) interpolation algorithms and improvements on those (typically $O(n)$ complexity), (2) is the class where still tractable $O(n^2), O(n^3)$ complexity yields high gains in performance, typically in orders of magnitude. Class (3) employs iterative (data-aided) expectation-maximization algorithms with $\geq O(n^3)$ complexity, which are typically unfeasible for implementation when considering the performance improvements that they yield and have a more theoretical value. We analyze and implement two multiuser WCDMA pilot-aided CE algorithms: *polynomial interpolation (PI)* (class 2) proposed by Yue et al. in [22], and *weighted multislot averaging (WMSA)* (class 1) proposed by Abeta et al. in [23].

Yue included comparisons between these in [22], showing that these two algorithms differ significantly in terms of performance, under multipath fading scenario. For single user single antenna systems, the bit error rate (BER) of PI is lower than that of WMSA over the whole. When considering bit energy to noise energy ratio (E_b/N_0) for single user antenna systems, the bit error rate (BER) of PI is lower than that of WMSA over the complete range, the difference exceeding an order of magnitude less when E_b/N_0 is greater than 6 dB. In case of multiple antenna, WMSA is outperformed by more than 2 orders of magnitude. The algorithm performance of PI stays superior for normalized Doppler frequencies in the range of $0.005 < f_d T < 0.013$ for both single and multiple antenna cases at an SNR of 8 dB. For multiuser systems, the performance of PI in RAKE receivers stays superior over that of WMSA over the whole range with the difference reaching 2 orders of magnitude when iterative interference cancellation is employed in medium to high SNR (>7 dB).

The rationale for this selection is the following: in the context of cognitive radio, wireless link state not always requires a class (2) algorithm performance, hence selecting a lower complexity class (1) algorithm could yield significant efficiency increase. Starting with an algorithm from classes (1) and (2) each, with major difference in their BER performance as well as in their complexity, we show that by exposing and exploiting the structural similarity, it is possible to design a flexible architecture which can adaptively switch among the two, without having the area overhead of two separate dedicated circuits. The architectural details are handled



Subtasks	Complexity order	Storage	Execution
Calculate α	$2N_s$ div, $6J$ add/sub	$2J \times N_p \times N_s$	Once
Average_pilots (1)	N_p add; 1 div	N_p	Every slot
Calc_estimate (2)	J mult /add	1	Every d_symb

FIGURE 1: WMSA details and complexity.

in later sections, after briefly describing computational traits of each of these algorithms.

3.1. Channel Estimation with WMSA. WMSA [23] is based on linear interpolation of known pilot symbols and has low computational complexity. For every kth user's lth path several N_p pilot symbols of slot m of the slot window are averaged first from received signal r_{kl} and initial estimate b_k:

$$\hat{\eta}_{kl} = \frac{1}{N_p} \sum_{n=1}^{N_p} r_{kl}(mN_s + n) b_k(mN_s + n), \tag{1}$$

where $l = \{1, \ldots, L\}$; $k = \{1, \ldots, K\}$; $n = \{1, \ldots, N_s - N_p\}$. The averaged values of several pilot symbols in a slot are weighted with precomputed coefficients α according to (2), to generate the estimates $\hat{g}_{kl}$ for each data symbol N_d, as Yue summarized it in [22]. The values of the coefficients α are thoroughly deduced and analyzed in [23]:

$$\hat{g}_{kl}\left(mN_s + N_p + n\right) = \sum_{j=-J+1}^{J} \alpha_j(n) \hat{\eta}_{kl}(m + j). \tag{2}$$

In Figure 1 the complexity of each of the subtasks of the algorithm is shown. The task for computing the α coefficients is executed once. The coefficient set can be changed based on WMSA algorithm parameterization and partially depends on the estimated symbol position. *Averaging* has to be done once per slot, in case there are several pilots in a slot. Then, for each data symbol, the estimate is calculated by summing the products between the averaged pilot value and the corresponding α coefficients of the symbol and slot. The dominant parameters of this algorithm from the complexity point of view are the size of the slot window $2J$ and the coefficients α. The larger the analyzed window, the greater the amount of needed storage. For the same J, storage needed for WMSA does not exceed that of polynomial estimation.

3.2. Channel Estimation with Polynomial Interpolation. The second algorithm is based on polynomial interpolation (PI) of the pilot symbols' channel values to approximate fading. As described in [22], the channel values are fit with a polynomial model of order q over $2J$ slots (3). Approximation is done by minimizing the mean-square error α based on the pilot symbols N_p in (4), N_s being the sum of N_p pilot and N_d data symbols. This translates to a Lagrangian interpolation problem, solved with (5) in (6), where $\boldsymbol{\eta}_{kl}$ represents the transpose of the pilot symbol vector constructed from $2J$ slots:

$$\begin{gathered}\hat{g}_{kl}((m+j)N_s) = \sum_{i=0}^{q} \alpha_i \psi_i(jN_s), \\ j = -J+1, \ldots, 0, \ldots, J, \\ \psi_i(n) \triangleq n_i; \quad i = 0, 1, \ldots q,\end{gathered} \tag{3}$$

$$\begin{gathered}\boldsymbol{\alpha} = \arg\min_{\alpha} \sum_{j=-J+1}^{J} \left[\hat{\eta}_{kl}(m+j) - \hat{g}_{kl}\left((m+j)N_s + \frac{N_p}{2}\right)\right]^2, \\ \text{where } \boldsymbol{\alpha} = \left[\alpha_0, \alpha_1, \ldots, \alpha_q\right]^T,\end{gathered} \tag{4}$$

$$\boldsymbol{\Psi} \triangleq \begin{bmatrix} 1 & (-J+1)N_s & \cdots & ((-J+1)N_s)^q \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & JN_s & \cdots & (JN_s)^q \end{bmatrix}_{2J \times (q+1)}, \tag{5}$$

$$\boldsymbol{\alpha} = \left(\boldsymbol{\Psi}^T \boldsymbol{\Psi}\right)^{-1} \boldsymbol{\Psi}^T \hat{\boldsymbol{\eta}}_{kl}(m). \tag{6}$$

Finally, we calculate the channel coefficients for data symbol n by using (7), the part of $\boldsymbol{\alpha}$ not depending on slot index m staying constant over the slot:

$$\begin{gathered}\hat{g}_{kl}(mN_s + n) = \psi(n)^T \boldsymbol{\alpha} \\ n = N_p + 1, \ldots, N_s \\ \psi(n) \triangleq \left[\psi_0(n), \ldots, \psi_{2J-1}(n)\right]^T.\end{gathered} \tag{7}$$

Subtasks/slot	Complexity order	Storage	Execution
Calc $\mathbf{\Psi}$ (4)	$\mathbf{q} \times \mathbf{2J}$ mult; $\mathbf{2J}$ add/sub	$(Q+1) \times 2J$	Once/slot
$\mathbf{\Psi}^T\mathbf{\Psi}$ in (5)	Matrix mult: $(\mathbf{q}+\mathbf{1}) \times \mathbf{2J}$	$(Q+1) \times 2J$	
Inversion in (5)		$(Q+1) \times 2J$	
Inversion subtask 1	$\mathbf{2J}$ div, $(\mathbf{q}+\mathbf{1}) \times \mathbf{2J}$ mul/sub	$+2J$	
Inversion subtask 2	$\mathbf{1}$ div, $(\mathbf{q}+\mathbf{1}) \times \mathbf{2J}$ div/sub, $(\mathbf{q}+\mathbf{1}) \times \mathbf{2J}$ mat·mult	$+2J$	$+2J$ times/slot
$(\mathbf{\Psi}^T\mathbf{\Psi})^{-1}\mathbf{\Psi}^T$ in (5)	Matrix mult: $(\mathbf{q}+\mathbf{1}) \times \mathbf{2J}$	$(Q+1) \times 2J$	Once/slot
Calc $\mathbf{\Psi}(n)$ in (7)	$\mathbf{q2} \times \mathbf{J}$ **mult**	$Q+1$	Once every data symbol
Calculate α (5)	Matrix with vector mult: $(\mathbf{q}+\mathbf{1}) \times \mathbf{2J}$	$(Q+1)$	
Get estimate (6)	Vector mult: $(\mathbf{q}+\mathbf{1})$	1	

Figure 2: PI details and complexity.

The performance and computational complexity of this algorithm depend on the polynomial order q and the analyzed window of $2J$ slots. Figure 2 shows the subtasks of this algorithm, their complexity, and storage requirements. The computational hot-spot contains matrix inversion and multiplication; therefore, the complexity rises steeply with the two main tunable parameters of the algorithm, q and J. Other parameters, such as number of pilots in a slot N_p, number of total symbols N_s, add additional flexibility to the algorithm. Most of the subtasks need to be recalculated for each slot and some simpler tasks (e.g., multiply-accumulates) are recalculated for every symbol. Having to multiply several matrices and performing matrix inversion for each slot makes this algorithm computationally very demanding. The trade-off range of parameters for the polynomial order is between 1 and 3, while the analyzed slot window $2J$ ranges from J equal to 1 to 4. Data dependency within the algorithm allows some of the storage to be reused, thus decreasing the demand on memory.

4. Related Work

This section is divided into two parts: the architectural background of SDR and existing implementations for WCDMA channel estimation.

4.1. Architectural Background. Over the years there have been several approaches to SDR architectures based on different architectural approaches. However, with the ever increasing complexity of new wireless standards a migration from flexible solutions towards clusters of inflexible ASICs can be observed.

Processors are flexible enough to implement complete standards. Architectures like SODA [3], EVP [4], and Imagine [24] tackle performance and power demands by employing high-speed vector processing or stream processing. In these architectures data parallelism is explicitly exploited. In SODA, a single instruction multiple data (SIMD) architecture is employed, where one ARM processor is coupled with 4 parallel processing elements. Tight control of bit width and bandwidth fulfilled power and performance requirements of two wireless standards. EVP takes a very long instruction word (VLIW) front-end to control several optimized SIMD units for specific SDR tasks, capable of handling multiple standards. Imagine is a media stream processor which has been retargeted for baseband processing in works like [25], exploiting clusters of parallel processing elements controlled by a host processor.

Application specific instruction-set processors (ASIPs) sacrifice flexibility in order to gain enough performance to tackle the more demanding applications from more recent algorithms. The FlexiChaP architecture [5, 9] customizes the pipeline, execution units, and data flow of a processor to accommodate convolutional, turbo, and LDPC decoding families, yielding an order of magnitude of speed-up compared with fully flexible processors like SODA.

Coarse-grained reconfigurable architectures like ADRES [26], RaPID [27], MorphoSys [28], RAW [29], Montium [11], and IMEC coarse-grained accelerator [6] employ arrays of data word level reconfigurable processing elements linked by a reconfigurable network, which can be tailored to a wider family of applications. Such coarse grained cores can cover a wide flexibility/performance range between ASIC and ASIP, like the application-specific FlexDet [10] or an ASIP-coupled rASIP [30]. Although this class promises SDR implementation capability [31], the difficulty in programming and exploring the design space of such architectures discourages wide-spread adoption.

System-on-chip solutions like Sandbridge [7] are increasingly popular, especially when high-performance scalable

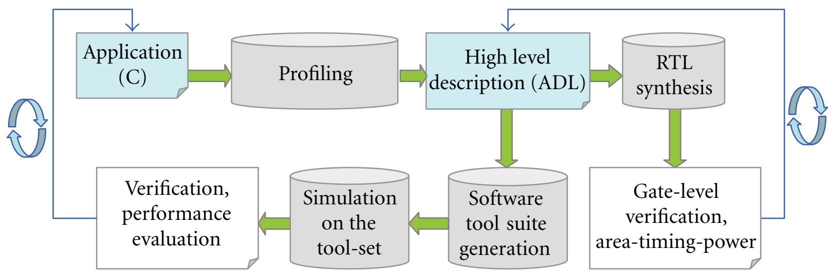

Application (C)
Profiling
High level description (ADL)
RTL synthesis
Gate-level verification, area-timing-power
Software tool suite generation
Simulation on the tool-set
Verification, performance evaluation

Figure 3: The standard LISA design flow.

ASIC cores [8] are employed to construct SDR components. Even hybrid approaches using accelerators and reconfigurable units are advocated [12, 32].

Field-programmable-gate-array- (FPGA-) based designs for SDR, like the WARP board, are extensively used for prototyping and research of new wireless standards and optimizations [33, 34], but power requirements make it prohibitive for end-products.

All these solutions except the ASIP/rASIP approach need "manual design" on either the hardware or the programming side or both. In this paper, our goal is to exploit off-shelf high level synthesis tool flow to generate (a) architectures with *just-enough* flexibility and (b) architectures with a coarse grained configurable core for greater flexibility. The fast exploration of flexibility and its effects on area and power from a commercially available tool flow differentiates our work from the existing approaches. Moreover, the proposed guidelines show that with minor extensions the tool flow can be exploited to generate also nonprocessor architectures.

4.2. WCDMA Implementations. Extensive research has been conducted by Rajagopal et al. to implement channel estimation and detection on stream processors and compare it with traditional DSP implementations [25, 35, 36] and also a VLSI implementation is conducted in [37], where area- and time-driven implementations are explored.

In case of the implementation on DSP, it is shown that the time required for computing channel estimation is 600 ms in case of 32 users [35], which is far too slow for real-time requirements. A dual-DSP and FPGA hybrid is shown to reach the real-time requirements for up to 7 users; however, the implementation also contains detection [36].

The implementation on the Imagine stream processor simulator shows major improvement over the DSP, but only the number of cycles could be extracted [25] which are at least an order of magnitude higher than the number of cycles reported in our work. Also, it is worth noting that the stream processor architecture uses 8 clusters of 3 adders and 3 multipliers which not only implies large area but also great power consumption. The computational hot-spots of matrix-matrix multiplication are implemented as a series of matrix-vector iterations which require a large number of cycles in the stream processor also due to data load/stores and movement.

For the VLSI implementation [37], the algorithm was analyzed and redesigned for efficiency, considering fixed/floating point representation trade-offs (up to 16 bits) and their effect on bit error rate; however, no direct comparison with our work could be made for several reasons: the design has not been synthesized, operating frequency is assumed, and area is expressed in terms of full adders, with no mention of storage. Additionally, no power consumption data has been reported.

5. High Level Design Methodology

In this section we present first the standard high-level synthesis tool flow of Synopsys Processor Designer, followed by the proposed guidelines and modifications for flexibility exploration.

5.1. High Level Synthesis and the LISA Language. The language for instruction-set architectures (LISA) is an architecture description language (ADL) which is used for modeling processors [20, 21]. This language is a high-level language with C-like constructs and is part of the commercially available Processor Designer tool-set from Synopsys. As shown in Figure 3, the design flow with LISA ADL allows generation of a synthesizable RTL description coupled with automatic generation of a set of tools such as C/C++ compiler, architectural simulator, assembler, and linker.

The flow starts with an application described in C, which is profiled to expose computational hot-spots and give insight about what kind of structures would be needed in the architecture. Usually, starting from a skeleton template processor, the architecture is described using the LISA ADL, which represents the main input to Processor Designer. This generates the tool-suite specially tailored to the architecture, like the simulator (step-by-step debugger), the compiler, assembler, and linker to run the application on the simulator.

Iterative design based on the performance evaluation allows incremental improvement on the LISA description, finishing with the synthesizable RTL generation once conditions are satisfied. If gate-level results are not satisfactory, the design exploration iterations can continue.

The LISA language is built upon a C-like syntax, with special structures to model timing and behavior of a processor; an example is shown in Listing 1. The `OPERATION` is the key element to describe the instruction set, timing, opcode, and behavior of the processor. This is a hierarchical construct, where several `OPERATION`s can describe one instruction partially, or mutually exclusive instructions within a tree (e.g., alu32 instructions). For example, `OPERATION alu32`

```
RESOURCE {
  REGISTER TClocked<int32> alu_in1, alu_in2, alu_out;
  PIPELINE pipe = {FETCH; DECODE; EXECUTE; WRITEBACK};
  /*...other resources */
}
OPERATION alu32 IN pipe.DECODE {
  DECLARE {
    GROUP opcode= { add || sub || and || or || xor };
    /*... other declarations of operands*/
  }
  CODING { 0b001 opcode source_operands }
  SYNTAX { opcode ~" " destination "," source_operands }
  BEHAVIOR {
    /* configure data path */
    OUT.dp_mode = opcode;
  }
  /* activate operand selection and datapath execution */
  ACTIVATION { source_operands, datapath }
}
//...other operations
OPERATION add IN pipe.EXECUTE {
  //... declarations of operand types, etc
  CODING { 0b00001 }
  SYNTAX { "add" }
  BEHAVIOR {
    alu_out = alu_in1 + alu_in2;
  }
}
```

Listing 1: Example LISA code.

can contain child operations like add or sub, which in turn can be parents to special cases like adding an immediate or a register. Parent OPERATIONs can activate its children via the ACTIVATION section. which assures correct timing across pipeline stages. Within this construct, arbitrary assembler syntax can be defined with SYNTAX, instruction encoding with CODING, and instruction behavior with BEHAVIOR, respectively. In the BEHAVIOR section, plain C code specifies the arbitrary functionality of the instruction and supports special data types such as bit[width] to allow close-to-hardware specification. The RESOURCE section is where global processor resources are defined such as memories, registers, and signals, along with pipelines, and pipeline registers. With these constructs, a processor can be fully described. For more information, please refer to [20].

5.2. Modeling Nonprocessor-Like Structures with LISA.

5.2. Modeling Nonprocessor-Like Structures with LISA. LISA is a powerful and flexible tool of modeling processors; however, due to the fine-grained instruction-based execution there is far more flexibility available in the design than required, which results in a performance decrease and energy inefficiency. We propose to reduce flexibility inherent in processors with architectures based on the following:

(i) *weakly programmable, barely flexible* structures, where processor data path is replaced by individual custom paths representing the data flow of the application, mimicking ASIC-like structures with exact amount of flexibility required;

(ii) *coarse-grained reconfigurable* structures which employ an array of word-size granularity, parallel, reconfigurable execution units with configurable interconnects for a greater amount of flexibility.

These two directions require *no modification* on the traditional LISA-based design flow, but they need a shift in how the data path is viewed, structured, and modeled. We propose modeling the application by use of *algorithmic state machine (ASM)* charts and *single qualifier double address (SQDA)* assembly, which expose the necessary information to model ASIC-like structures, as well as coarse-grained reconfigurable arrays. The proposed modeling steps replace profiling in the traditional LISA tool flow (Figure 4). The ASM chart exposes necessary hardware structures to the designer from which the hardware description can be deduced, while the SQDA assembly captures application behavior and translates it into executable binary with the help of the automatically generated assembler and linker.

5.2.1. Modeling the Application with ASM Charts. Algorithmic state machine charts are commonly used for specifying digital circuits. They have been proposed for use in HLS earlier [38, 39] and also an ASM specification language has been proposed [39], along with extensions, such as

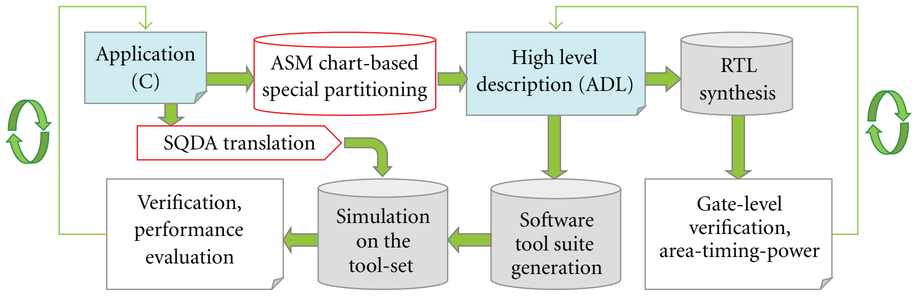

Application (C)
ASM chart-based special partitioning
High level description (ADL)
RTL synthesis
SQDA translation
Verification, performance evaluation
Simulation on the tool-set
Software tool suite generation
Gate-level verification, area-timing-power

Figure 4: LISA design flow with the proposed enhancements.

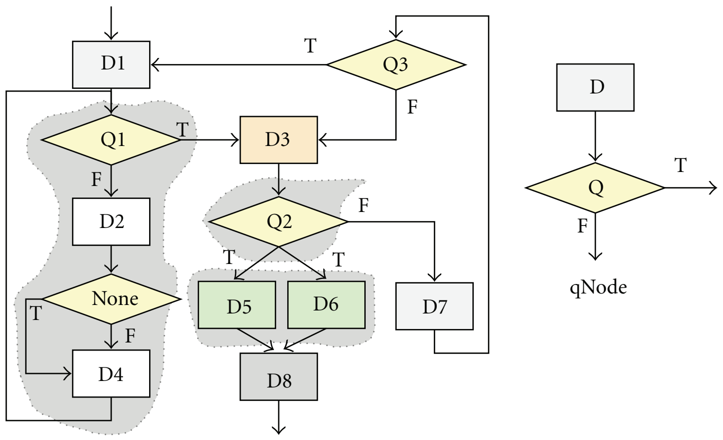


Figure 5: Example of ASM chart with partitions, and the 3-tuple called qNode.

communication channels and user-defined operators. ASM charts provide options both at algorithm and architectural level for adding or removing flexibility to gain the needed amount of weak programmability to accommodate the application.

The ASM graph isolates the parts of the application which are executed conditionally, shows the range of loop iterations, and identifies possible parallel execution. The graph can be constructed from the C implementation of the application in a straightforward way, which in itself prepartitions the application (functions, loops, if-else statements, etc). It is structured in a 3-tuple (D, Q, E) in which the source statements can be grouped into *data path nodes (D)*, *qualifier nodes (Q)*, and *edges (E)* connecting them. Figure 5 shows a part of such a coarse graph and shows further possibilities for finer partitioning.

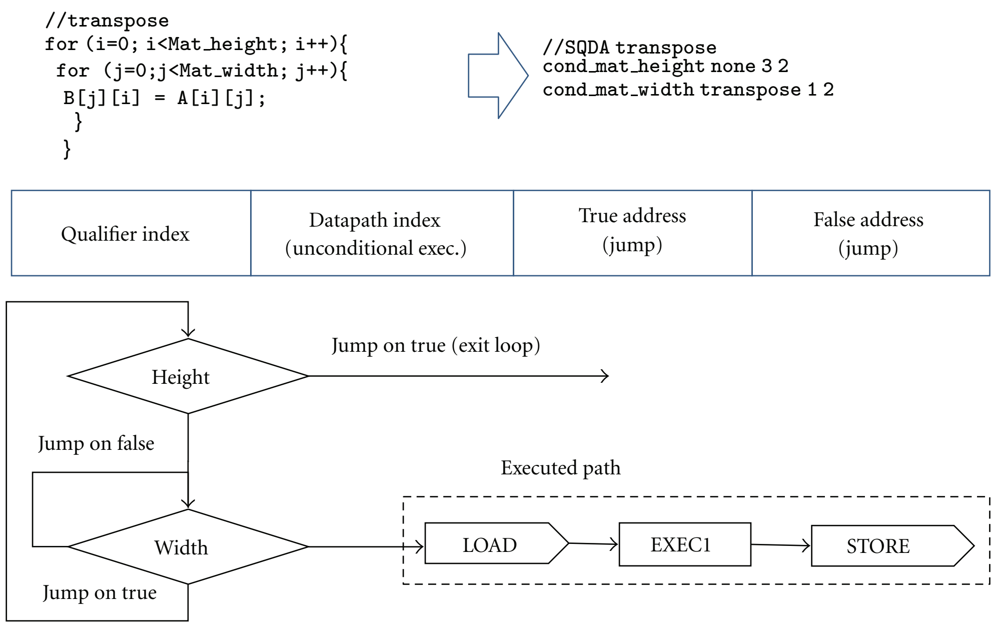
When targeting ASIC-like behavior, the ASM chart is divided into two groups: *control* and *data execution*. The control group contains all the qualifier nodes (Q-nodes), which represent conditional execution statements, branching, and (un)conditional jumps within the application. The data execution groups contain all the data computation nodes (D-nodes) exposing parallelism and the actual required computation type and complexity.

The designer can choose to group Q and D nodes into a larger group, which specializes the data path or can reduce group size down to elementary operations to allow more commonalities between groups, which adds flexibility of implementation, resulting in trade-offs of speed, area and power for flexibility. Parameterization of larger groups allows a finer degree of flexibility control, specifying exactly what parts are shared and which need custom data paths in hardware. Each group of D-nodes can be directly coded in LISA for custom ASIC-like execution paths. When combining the Q-nodes, a finite state machine results, each Q group representing a state, effectively controlling which D group is activated at what time.

For instance, in Figure 5, node Q1 represents a state in which the data group made of D2, an empty Q-node, and D4 is repeatedly executed until a condition is fulfilled, then D3 will be activated. The succession of D2 and D4 can be implemented as one custom data path (less flexible) or two separate data paths (more flexible allowing reuse of D2 or D4).

The proposed use of ASM charts as an intermediate layer in the design flow not only enhances the designer's insight to what hardware will be needed for the application, but also helps mapping the application to the hardware.

Our experiments show that this process can be automated. The ASM chart is used as an intermediate representation between the C source code and LISA description allowing generation of all necessary intermediate files like program assembly, ADL description. The automation of this process completes the flow to provide a customizable C-to-RTL flow completely under the control of the designer. For details on the automatic generation and optimization of



```
//transpose
for (i=0; i<Mat_height; i++){
 for (j=0;j<Mat_width; j++){
  B[j][i] = A[i][j];
  }
 }
```

```
//SQDA transpose
cond_mat_height none 3 2
cond_mat_width transpose 1 2
```

Figure 6: SQDA and how it works on the example C code.

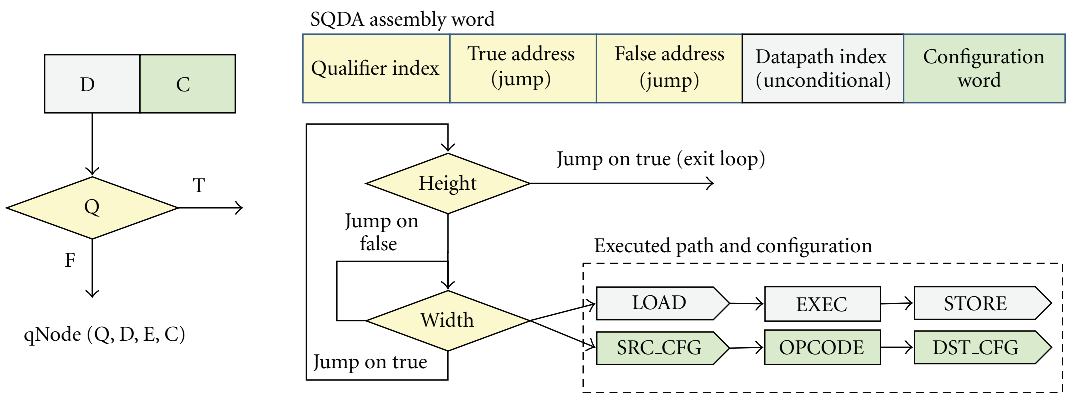


Figure 7: Adding configuration data to the qNode and SQDA to support coarse-grained reconfigurability.

ASM charts and the generation of respective LISA descriptions for ASIC synthesis we would kindly refer the reader to our recent paper [40].

5.2.2. The SQDA Assembly. The single qualifier double address (SQDA) assembly is used to abstract the Q and D node pairs into a form that can be interpreted by an assembler. SQDA needs to convey which Q-node activates which D-nodes and what happens when the Q-node evaluates to true or false, effectively encoding the edges (E) of the ASM chart. SQDA is composed of 4 fields: the *qualifier* field encodes the ID of the current Q-node, while the *executed path* represents the D-nodes linked to the Q-node and the two addresses pointing to the next qualifier in the execution path, just like the edges E.

The example in Figure 6 shows the C code of a nested loop, its SQDA translation, and the execution chart. SQDA statement 1 has no "false" data path and goes directly to statement 2 when false and exits the loop when true, executing a conditional jump. Statement 2 keeps the state, repeatedly firing `transpose` data path as long as `cond_mat_width` evaluates to false then exits the inner iteration by going to statement 1. The *true* and *false* jumps are addresses in the program memory and can contain a self-reference, effectively creating a *state* or can represent (un)conditional jumps to other addresses implementing *control flow*. Using such a construct, the complete application can be written in a compact form, but also it can be parsed by the assembler generated from the LISA description.

5.2.3. Extending ASM and SQDA for Coarse-Grained Reconfigurability. With minor modifications, description of reconfigurable structures is also possible using ASM and SQDA. The additional flexibility gained from using coarse-grained reconfigurable structures enhances resource share, execution concurrency, and resource repurposing with a finer control on how different applications map to hardware; however, it is often difficult to describe and deduce the necessary configuration bits. Especially, mapping of a given application to a reconfigurable structure poses a difficult challenge.

To easily code reconfigurability in LISA and keep track of necessary configurations, we propose extending the ASM and SQDA of the qNode tuple (Q, D, E) with a *configuration node* (or C node), which is strongly coupled with a D-node in the chart, as shown in Figure 7. The C nodes must contain the configurations required to recreate the D-node and Q-node equivalence in the reconfigurable structure. This process is straightforward, because the D node contains the necessary information about execution type that can be mapped to one or more coarse-grained elements, along with the source and destination routing information based on data locality information of the edges E from/to that D node.

Similarly, the SQDA assembly must also contain the configuration term (Figure 7) so that D and C nodes are activated simultaneously with one assembly instruction. This allows controlling of all configurable parts just by entering the necessary configuration word in the application assembly code, bringing the configuration information to assembly level.

The standard LISA language has enough flexibility to describe reconfigurable structures by exploiting the template constructs for similar resources and operations. These constructs are similar to C++ templates, with the restriction that template variables have to be constant at compile time. For instance, when describing 8 identical processing elements, only one element needs to be described in a templated form (e.g., `element⟨0⟩` and `element⟨7⟩` are different (and static) instances of the template `OPERATION element⟨id⟩`). Similarly, routing interconnect can be described.

These steps enable simulation and debugging of the reconfigurable core using the LISA design flow: assembler for SQDA is generated automatically and the LISA debugger can be used to simulate execution of the complete architecture.

Moreover, since the configuration data in the C nodes is closely related to the D node contents, the configuration bits can be generated internally by the D node, without exposing the configuration bits to assembly level. Essentially using the ASM chart information, a CGRA can be constructed which self-generates its configuration bits just by taking one short instruction. This reduces complexity of the assembly code and the SQDA assembly of the application can remain unmodified; however, flexibility is lost because certain configuration combinations are available only when the instruction corresponding to the D node which generates them is activated. In this work, full flexibility was exploited and configuration words were provided via assembly code.

6. Architectural Exploration

From the architectural point of view, implementing the target algorithms is challenging because of the mixed internal computational components: matrix inversion is a sequential process where control flow dominates (partial pivoting and backward substitution), while matrix multiplication is a task where much parallelism is available. An architecture that executes both very efficiently must be considered. Two architectural classes are explored based on using the methodology described in previous sections.

(i) Architecture 1. *Just-enough flexibility* is embedded to support the control-flow dominant part and several execution units are used to execute the parallel parts of the algorithms efficiently.

(ii) Architecture 2. *A coarse-grained reconfigurable core* is employed, which enhances parallel execution and flexibility but keeps the flexible SQDA control-flow front-end for the sequential parts.

Both architectures are implemented using fixed-point arithmetic in the Q-format for 32 and 64-bits (Q16.15, Q33.30).

6.1. Arch. 1: ASM-Based Design with Minimal Flexibility.

In order to find the minimum flexibility required to support both algorithms in an efficient manner, a minimalistic step-by-step construction approach of the architecture is performed, starting from the ASM chart of both target algorithms.

Partitioning. For the data path partitioning, first the basic data path nodes from the ASM chart need to be identified. Shared data path nodes and those that require flexibility are individually modeled in LISA. Otherwise, the data path nodes are merged with the preceding/succeeding data path node. Once the complete algorithms are described individually using ASM charts, identification of common resources and paths can be done. The aim is to reuse as many resources as possible across the two algorithms. An example partitioning is shown in Figure 8, taking a piece of the ASM charts of both algorithms, partitioning the nodes and identifying common points. This step is essential for sharing resources and creating parameterized custom D-node paths which can support both algorithms.

Control Flow. The control flow essentially requires support for if-else statement and loop statements. From the ASM chart description, the control statements are identified which require independent tuning. Otherwise, the if-else statements are merged within a larger data path. Various step sizes for the loop iterators are supported, allowing shared structures across different loops. In the control-flow implementation via SQDA, the program counter always jumps to the new instruction address for both true and false outcomes of the conditional checks. Program counter jumps should not incur delay penalties; therefore, the condition check and instruction fetch are part of the first pipeline stage. To allow a direct coupling between the Q-node and D-nodes, they have to be part of the same assembly instruction word. Due to a limited number of assembly instructions required to program the architecture, the program memory is quite small in size (640 bits). Therefore, the instructions can be conveniently stored in register files constructed out of standard cell memories (SCMs) offering fast asynchronous access.

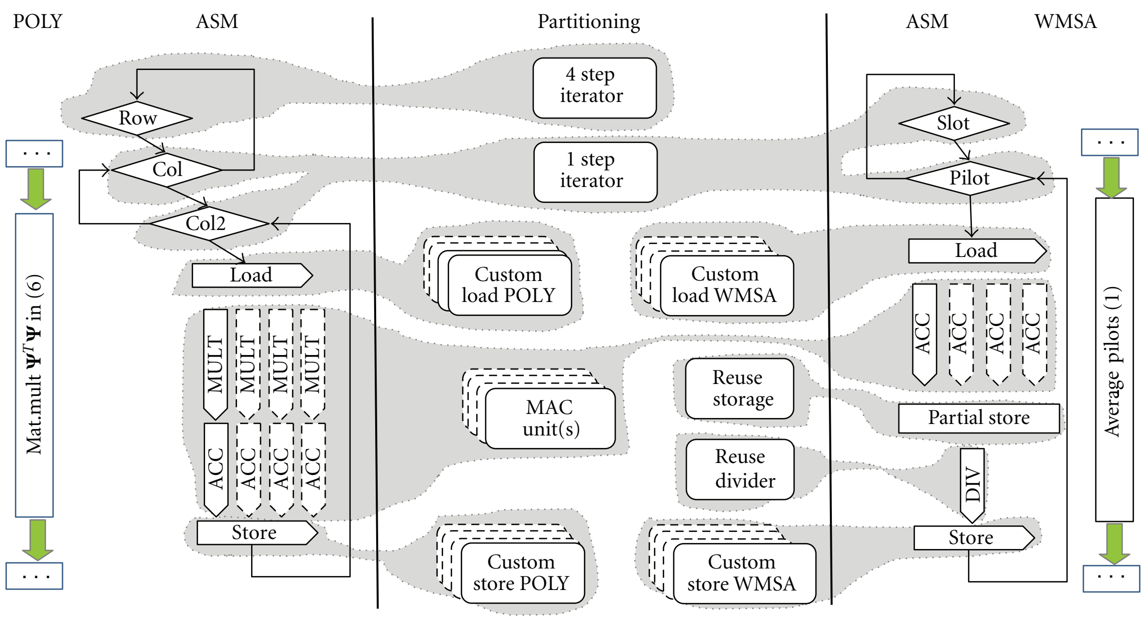


Figure 8: Partitioning of a piece of the ASM chart of the applications, reusing common points across applications and exploiting parallelism.

Load/Store and Pre-/Postprocessing. Analyzing the amounts of data needed by the subtasks and usage patterns, optimal load operations become coarse-grained operations composed of multiple memory accesses, shuffling and selection of data. This is problematic for an efficient implementation with SRAMs with limited number of ports. We considered an SCM-based implementation, which offers asynchronous read and synchronous writes. This takes a heavy toll on area but reveals the maximum possible runtime performance. Therefore, data access addresses can be hard-coded for each load/store pattern of an execution node, bundling the complete load/store processes in sets of patterns tuned for the respective data path. This not only renders data fetch address computation unnecessary, but also avoids memory operations like matrix transpose.

Execution. Looking closer at the flexibility requirements, that is, what kind of parameters the applications have and how they influence the amount of processing, we can link the complexity to the required execution resources. For PI, the width of the matrix depends on the polynomial order +1, while the height depends on $2J$ slots considered as observation window. A typical value for the polynomial order is 3 and for the slot window J is 2 and can change by factors of 2, yielding matrix operations of matrix size of 4×2, 4×4, 4×8. A flexible number of multiply-accumulate (MAC) units allow flexibility exploration, so the design is easily adaptable to energy, area, and timing needs. Thus, when using 4 MACs, a 4×4 matrix multiplication can be done in 16 cycles. Parameter changes result in a different number of iterations, which translates in different counter increments in the control path. These units can be shared among subtasks which use matrix multiplication and can accommodate other multiplications and additions as needed. In matrix inversion, division is also needed, so one divider completes the minimum set of execution units. It must be noted that these MAC units and the divider are executing all operations in Q-format calculus, meaning that each multiplication or division is a concatenation of operations packed into one unit. While simplifying architectural description and programming, this causes a very long critical path in the design.

The Resulting A1 Architecture. Managing to partition the application into the qualifier (Q node), load (D node preprocessing), execute (D-node computation), and store (D-node postprocessing) the architecture became inherently pipeline-able through 4 stages (SA, LD, EX, ST), shown in Figure 9. Pipelines are inherently supported by LISA, generating control and pipeline registers automatically.

The SA stage takes care of instruction fetch, qualifier evaluation and activates the respective data-path in the next cycle. The data path associated with this instruction is activated in the next stage (next cycle). The LD/ST stages contain memory access pattern sets, activated by the SA stage and properly timed, thus loading/storing relevant operands to/from execution unit input/output registers. Some LD/ST patterns are parameterized, yielding different data for qualifiers in different states. Compared with the control logic and LD/ST pattern sets, execution units are much larger, even for fixed point arithmetic. To reduce the area impact, these units are shared across a large number of ASM D-nodes by statically linking the units to the input and output pipeline registers. By that, area consumption is increased in form of multiplexing the data from register files to the input/output points of MAC/DIV units. Data dependency is avoided by performing data forwarding between EX, ST, and LD stages. With most structures shared among the algorithm, few

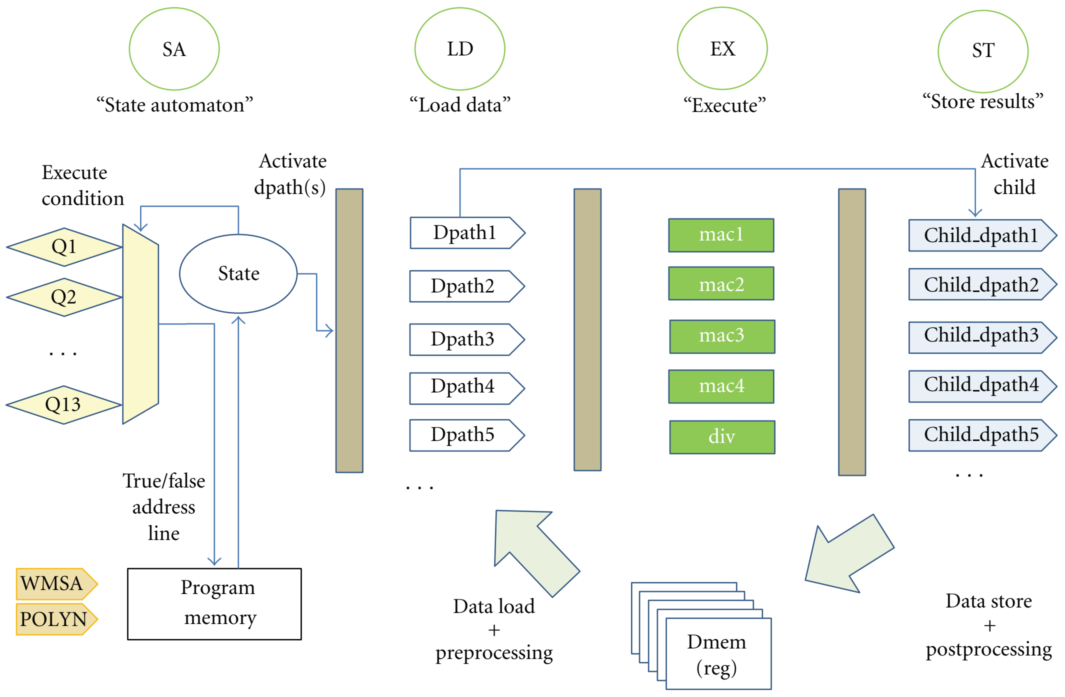


FIGURE 9: The architecture with *just-enough* flexibility to support the two channel estimation algorithms.

algorithm-specific structures remain in the form of specific LD and ST operations, which translate in little area overhead.

6.2. Arch. 2: Coarse-Grained Reconfigurable Core

Considerations. To explore the effect of a flexible design, allowing a finer control of efficient execution, the MAC units are replaced by a reconfigurable core. The reason for exploring such a structure is the fact that the Q-format MAC units and DIV unit are extremely large and have long critical path, as noted in the results section. Some operations, especially in WMSA, use the large MAC unit although only plain addition or subtraction is needed. Also, in the LD/ST pre- and postprocessing stage some data paths included additional adders and shifters, to accommodate required processing, which resulted in increased energy needed per symbol. By using a reconfigurable core, we aim to split the critical path of the MAC units, while making pre- and postprocessing tasks more efficient by using structures more suited for the task, configuring each D-node with exactly the required structures for its processing requirements. Therefore, the processing element (PE) configuration is tailored to execute Q-format arithmetic efficiently.

Partitioning. Except for the execution part, the considerations for partitioning, control-flow and load/store processing from the previous subsection apply also for such an architecture. The configurable core is defined using the extended ASM modeling from the previous section.

The Resulting A2 Architecture. As shown in Figure 10, the pipeline structure is heavily modified; now only 2 pipeline stages remain, one for control (SA) and one for execution (EX). The structure of the core in (EX) is based on expanding the large Q-format MAC units from the previous architecture giving a heterogeneous structure (Figure 10): the first column (PE0, 4, 8, 12) contains also multipliers beside adders and shifters, the second and third columns contain adders and shifters of double bit width ($2 \times Q$, $Q = 32, 64$) while the last column contains Q bit width adders and shifters. The divider is added with a direct connection to the outputs of elements 5 and 9, which are the endpoints when computing the shifting and adding from Q-format division. A mesh structure is sufficient to link these structures together and link to the load/store patterns from/to memory. It must be noted that, due to the way Q-format arithmetic works, intermediate results within a multiplication or division are of double bit width ($2\times Q$), hence when forwarding results from columns 0 to 1, 1 to 2, the interconnect must accommodate double bit width $2 \times Q$.

Reconfigurability and Interconnect. One processing element can take input data from 12 sources, 6 for each input, as shown in Figure 10. Besides the 4 neighboring PEs (north, south, east, west), one PE can also take its own output register as the source or take data from memory. The wires for each are directly connected to the output registers of the respective neighboring nodes (essentially creating a 2D pipeline within the array), or a certain load pattern in case of the memory links. PEs on the border of the array take the load pattern as a source, when a neighbor in that direction does not exist (e.g., PE0 north and west links are the same with the memory link). The output registers are also directly connected to certain store patterns. Depending on the application, the load/store patterns may or may not connect to each PE, as

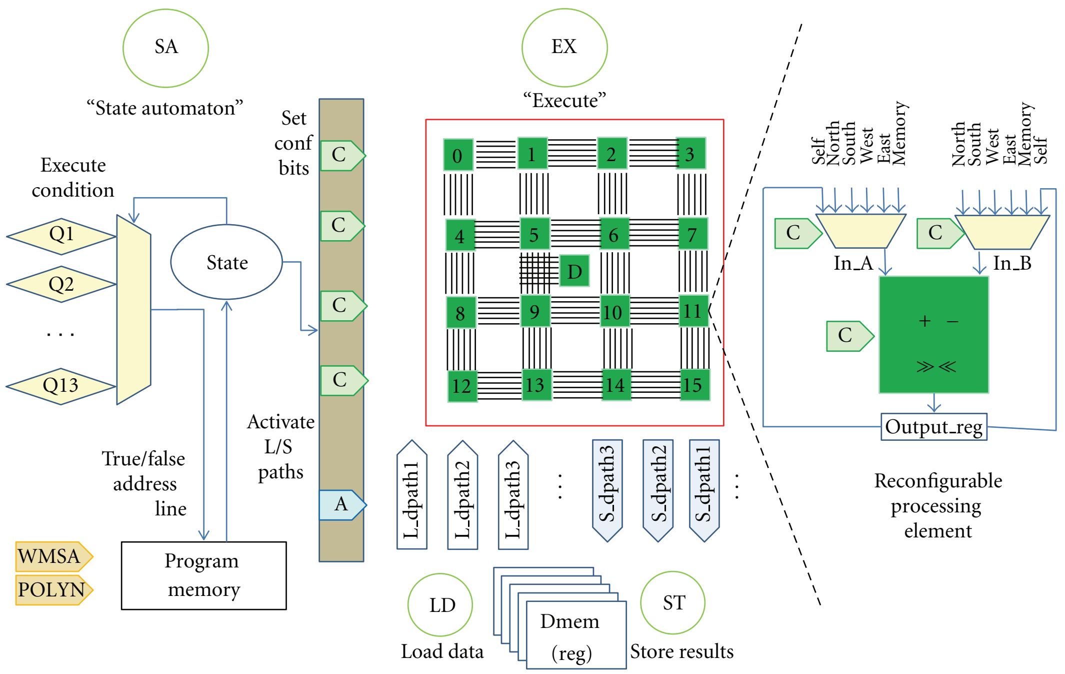


Figure 10: The coarse-grained reconfigurable core-based flexible architecture: 2-stage main pipeline with SA and EX, 4 × 4 reconfigurable core with mesh interconnect, and a zoom-in to one of the elements of the core.

the data can be processed by a PE chain before being ready for storage.

There are 3 configuration bits for each source multiplexer, and 3 bits for the opcode selector. In LISA this is described as a template `OPERATION⟨⟩` for each element, which takes the data on input wires `in_A` and `in_B` and outputs to the output register, considering the opcode for the execution. The 6 interconnect links for each source are modeled also with template `OPERATION⟨⟩`s which are activated based on the configuration bits. Thus, to completely configure one element, 9 configuration bits are needed, resulting in 144 configuration bits for the entire reconfigurable core of 4 × 4 elements. These bits are stored directly in the pipeline register by the SA stage, right after reading the instruction word.

Programming View and Mapping. The instruction word contains also the decoding bits of the load/store patterns (instructions), 6 bits for each, enabling a maximum of 128 load and store patterns. The SA qualifiers are replaced by five *timers*, essentially configurable counters which decide for how many cycles one SQDA instruction word holds true. The complete SQDA instruction word holds thus qualifier encoding (4 bits) immediate true and false addresses (2 × 8 bits), the load and store encoding (2×6 bits) and the configuration word (144 bits) resulting in a 176-bit instruction word, closely following the modified SQDA specified in the ASM-based flow.

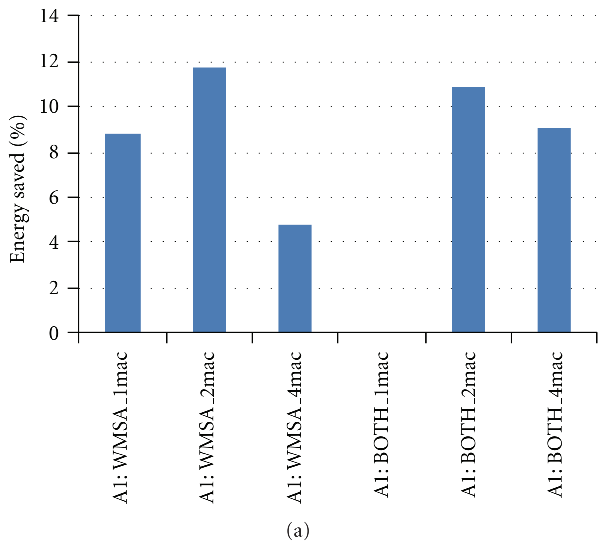
Mapping the application from the ASM chart is done in the most efficient way allowed by the core's flexibility. The C node of one data path is translated into the source A and B configuration bits, and the D node contents into the opcode. When one D node was created by merging smaller D nodes, it will be mapped to several PEs spatially and/or temporally based on input/output locations, dependencies, or available PEs. This will create SQDA instruction words with a combination of qualifier/no qualifier, load/no load, store/no store, configuration word or empty configuration (NOP, the array is not used). The load/store patterns are now exclusively signal assignments from/to storage to/from input/output, without any data processing, as all data processing happens inside the array. Some of these can control qualifier status (reset/set the *timers*).

7. Evaluation

We synthesized the generated RTL descriptions of 18 different design points for Architecture 1 (A1), 9 design points for Architecture 2 (A2) using Synopsys DC D-2010-SP3. For all designs, *Faraday* 90 nm standard cell technology library was targeted, with clock-gating and operand isolation power optimizations enabled and annotating RTL switching activity for Power Compiler. Since synthesizable RTL code is generated by the tool flow, FPGA-based implementations may also be targeted; however, standard cell library was chosen to provide a clearer comparison for area and power.

A1 template required 2 k lines of code in LISA for 61 operations and has 40 k lines of generated Verilog code. A2 has a larger LISA description of 4.4 k lines for a total of 98 operation instances (expanding the templates), which generates 51 k lines of Verilog code.

In A1, partitioning of PI with $J = 2$ and $q = 3$ yielded 12 qualifiers and 34 data access patterns, requiring 39 SQDA instructions. WMSA partitioning with $N_p = 2$ and $J = 2$ resulted in 1 extra qualifier and 17 data access patterns, coded in 13 SQDA instructions.



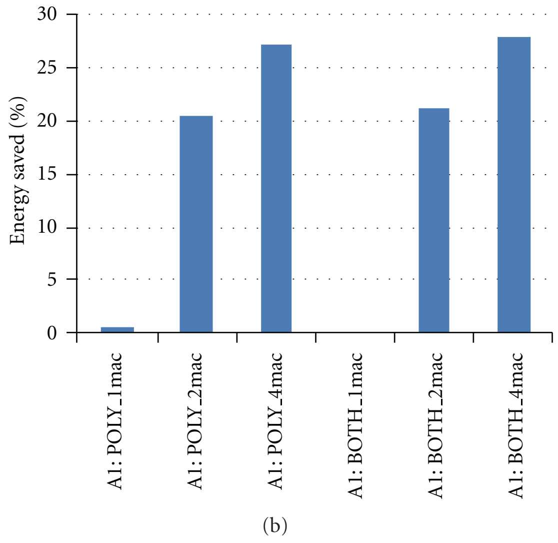
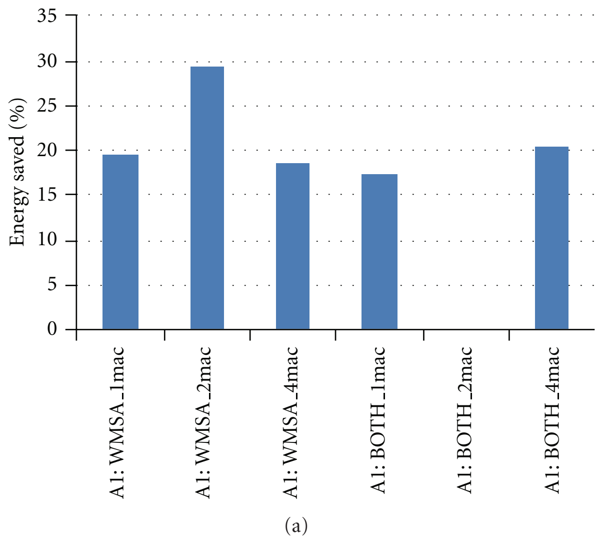


Figure 11: 32-bit: dedicated A1 versus hybrid A1 energy consumption normalized to worst case. WMSA (a) and PI (b).



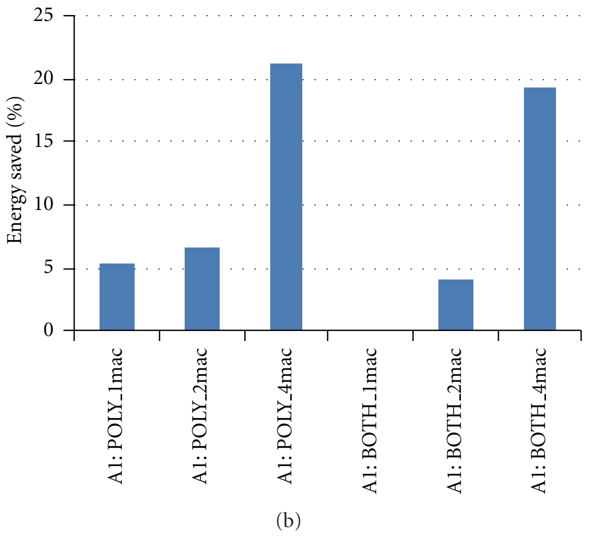


Figure 12: 64-bit: dedicated A1 versus hybrid A1 energy consumption normalized to worst case. WMSA (a) and PI (b).

In the case of A2, with the same algorithmic parameters, polynomial mapping yielded 36 access patterns, 7 qualifiers and has an assembly program of 102 instructions. WMSA yielded 16 access patterns with the same qualifiers (reconfigured) and 50 SQDA instructions.

First, design points across an architectural class are compared, then cross-class comparisons are presented with the following convention for the graphs:

`arch.class":"algorithm"_"design point specialty`

For instance, `A1 : wmsa_1mac` means first architecture class ("just-enough" flexibility), supporting only WMSA and having only one MAC unit, while `A2 : both_25` means second architecture class (coarse-grained core) supporting both algorithms, running at 25 MHz.

All results are for a complete algorithm execution for a slot of 10 symbols for the respective application. Slot structure was comprised of 2 pilot symbols and 8 data symbols, while $J = 2$ for both algorithms, $q = 3$ for polynomial.

7.1. Intra-Architecture Class Design Point Comparison

7.1.1. Architecture 1. A1 design points resulted from combining 1, 2, or 4 MAC units, and 32/64-bit versions for dedicated structures for one algorithm, then for the hybrid architecture to explore the design for low power or low energy, while maintaining minimum flexibility. Comparing the dedicated design points for one algorithm with the ones supporting both, it can be noted that for A1, when running PI, the combined flexible architecture supporting both algorithms comes close ($\leq$5%) to the energy per symbol of the respective dedicated architecture as shown in Figures 11 and 12. When the combined architecture is running WMSA, it

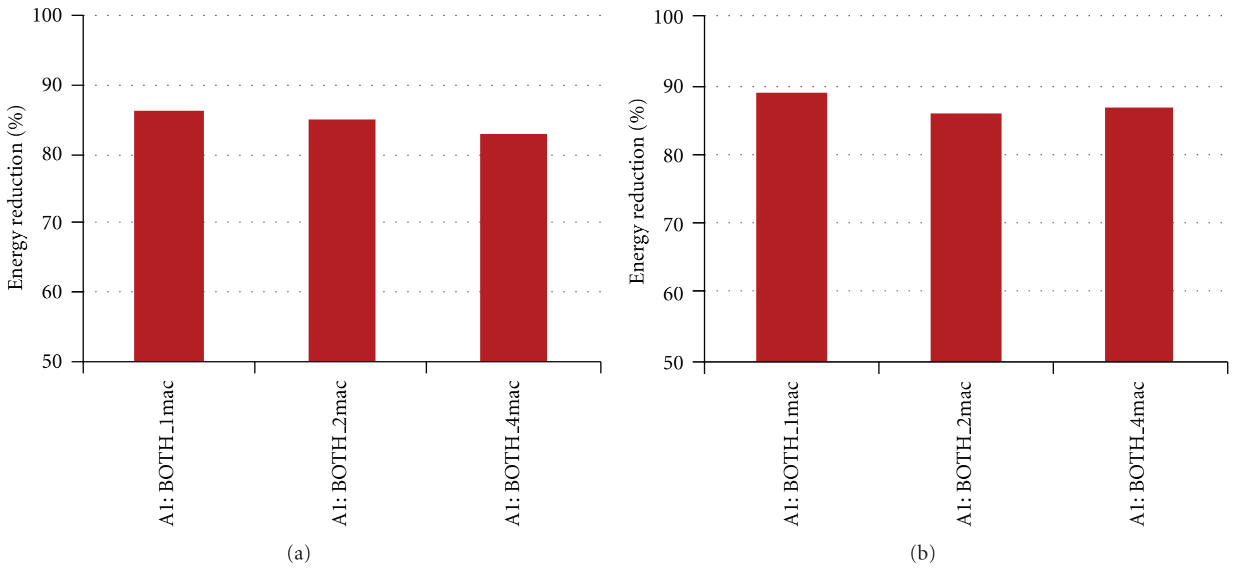


Figure 13: Energy savings in percent for hybrid A1, during adaptive switching (32-bit (a), 64-bit (b)).

uses comparable or less energy ($-5 \sim 8\%$) compared to the dedicated WMSA architecture, explained by the fact that with some partitioning, WMSA may be executed more efficiently on the structures for PI, leading to energy saving without much overhead. For all design points, the Q-format divider was the timing bottleneck, limiting frequency to 25 MHz. It must be noted that in the implementation, one can easily switch between these design points in order to seamlessly trade off performance against energy, power, or runtime.

The architectural flexibility can be utilized within one algorithm or across algorithms depending on performance constraints. Varying execution unit count yields up to 20% energy and up to 50% power savings for PI, while for WMSA up to 14% energy and up to 38% power can be saved. Figures 11 and 12 show the relative energy per symbol difference across design points for one application. For WMSA the values are within 12%; however, the dedicated 2 MAC architecture is most efficient. For polynomial, both the dedicated and the combined ones have similar values.

The execution time per slot ranges between 31–78 μsec for PI and 7–11 μsec for WMSA, while WCDMA hard deadline is 670 μsec, allowing extra savings by frequency scaling, and so forth.

Figure 13 illustrates how much energy is saved when adapting to better signal conditions by switching between the two algorithms: 10–41% *power* and 81–88% *energy*. The savings stay consistent across design points. For switching between the algorithms, one only needs to load the respective instructions from the program memory. On top of this, the algorithms can be adapted further internally by fine-tuning the points typical of WMSA and PI.

For area critical situations, the architecture template can be easily retargeted, for example, the 1 MAC unit design executes in double number of execution cycles of the 4 MAC design, but saves 36% area. The area difference between the dedicated PI architecture and the combined one over the design points is between 5.69% and 12.8%, which is negligible when compared with the joint area overhead of two dedicated structures (205%–212%), even more so when considering the energy savings.

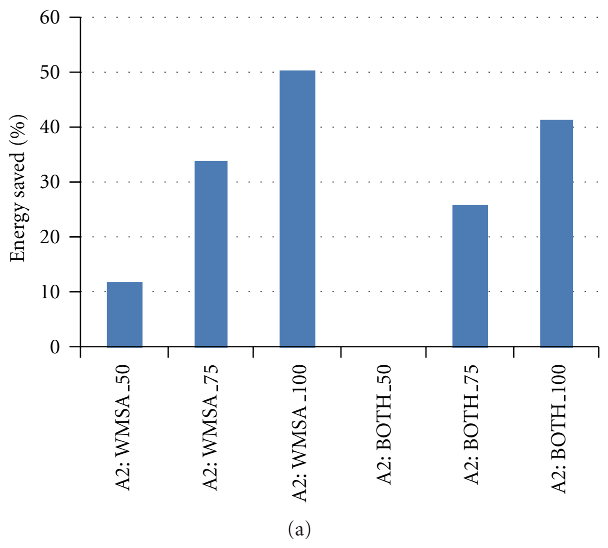
Unfortunately, we could not compare our results with the existing implementation in [37] due to the reasons stated in Section 4.

7.1.2. Architecture 2. A2 design points have been constructed from 32/64-bit versions of the architecture tailored for each algorithm and the hybrid version. To further analyze how the coarse-grained core affects energy, we synthesized for different frequencies, showing that frequency closely affects the energy per symbol value. When the frequency is low, the algorithms take longer time to finish, even if they have lower total power; consequently, the resulting energy value is high. Near critical-path operation severely impacts power consumption and area. This is especially the case in the 64-bit design (Figure 15).

Different mapping choices did not affect the power values, since the employed mapping strategy was to use the closest element which matches the needed operation. During mapping, no congestion was observed, for two reasons: (1) ASM chart-based mapping separates the execution in independent, parallel threads and not many threads are competing for the same processing element (except the divider in WMSA for the filter coefficient calculus); (2) the matrix inversion processing is sequential, the data dependencies of the inner loops limit parallelization in a natural way, while matrix multiplication exploits 100% array usage in a well-defined manner.

We have compared the results in a similar way as for A1, shown in Figures 14 and 15 detailed as follows:

(i) 32-bit WMSA (Figure 14(a)): as frequency increases, the energy reductions scale linearly with throughput;



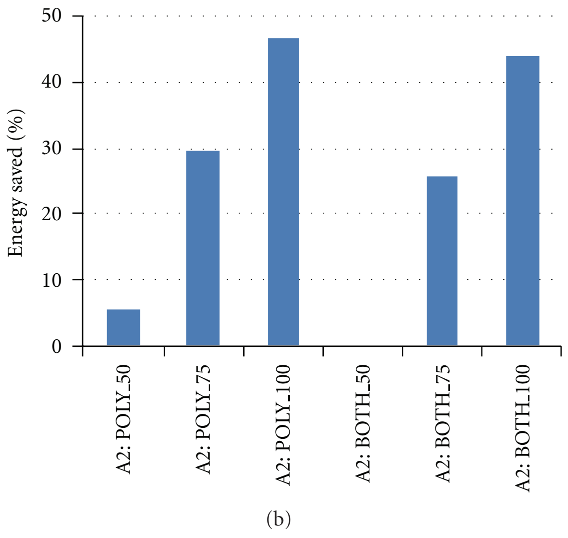
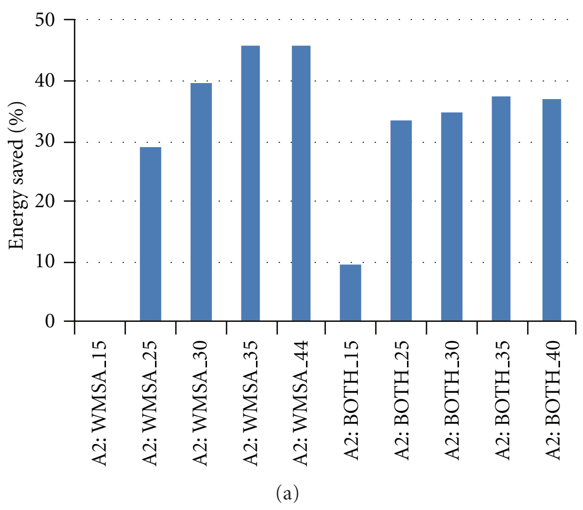


FIGURE 14: 32-bit: dedicated A2 versus hybrid A2 energy consumption normalized to worst case. WMSA (a) and PI (b).



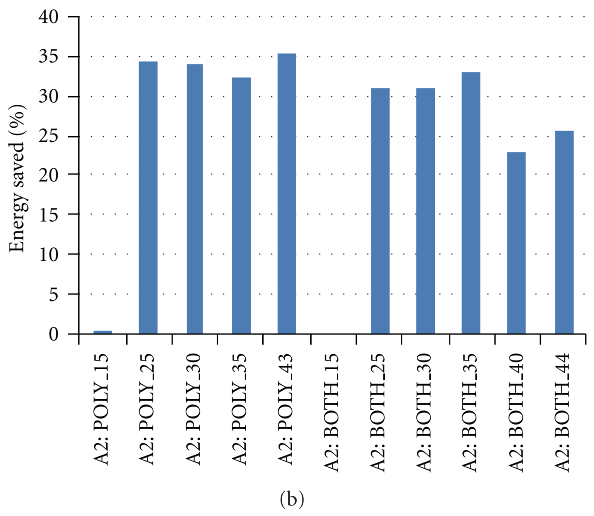


FIGURE 15: 64-bit: dedicated A2 versus hybrid A2 energy consumption normalized to worst case. WMSA (a) and PI (b).

however, the combined architecture has around 10% lower energy savings compared with the dedicated structure,

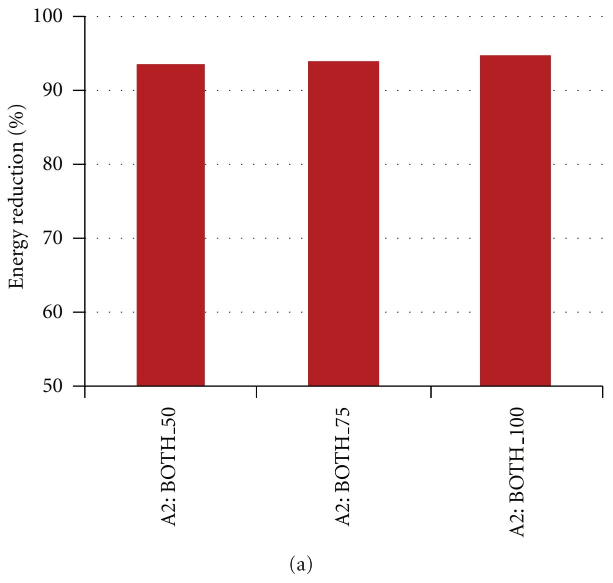
(ii) 32-bit PI (Figure 14(b)): energy reductions are similar; the difference is only around 5%,

(iii) 64-bit WMSA (Figure 15(a)): due to the high power consumption as frequency increases, the increased throughput cannot compensate enough, and the energy saving hits a limit as maximum frequency is reached. Also the combined architecture fares 10% worse when compared with the dedicated architecture when it comes to energy savings,

(iv) 64-bit PI (Figure 15(b)): when scaling frequency there is an inflexion point, where best energy savings are attained and for which similar savings of the dedicated structure can be reached (within 5%). Similar to WMSA, near maximum operating frequency the savings diminish (10% difference).

For the scenario that A2 adapts to better signal conditions, 20–44% *power reduction* and more than 93–97% *energy reduction* can be attained, as shown in Figure 16. Adaptation is similar to A1: only the respective assembly program needs to be executed for a switch from PI to WMSA. However, a higher flexibility allows for a finer control and usage of the structures, improving energy save.

7.2. Inter-Architecture Class Comparison and Discussion. When comparing across architectures the first point to note is the significant difference in area (Figures 20 and 21). Due to the use of the coarse-grained core, A2 has almost always a larger area than A1 at comparable design points. This can be explained by the fact that due to the Q-format arithmetic, the



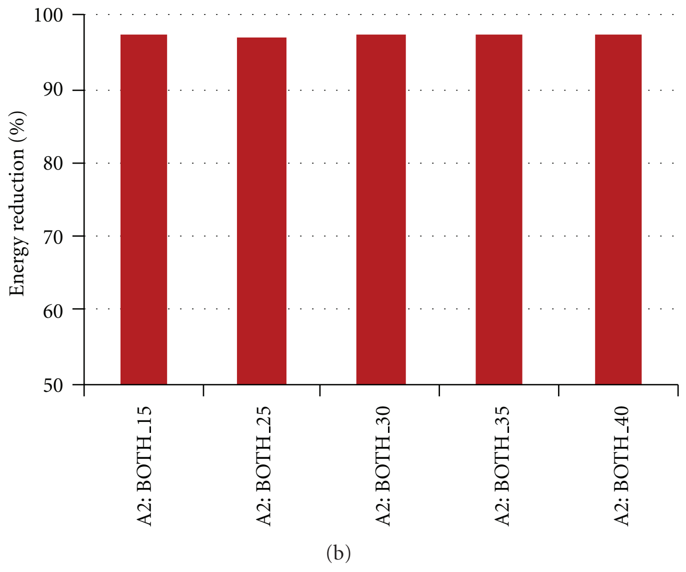
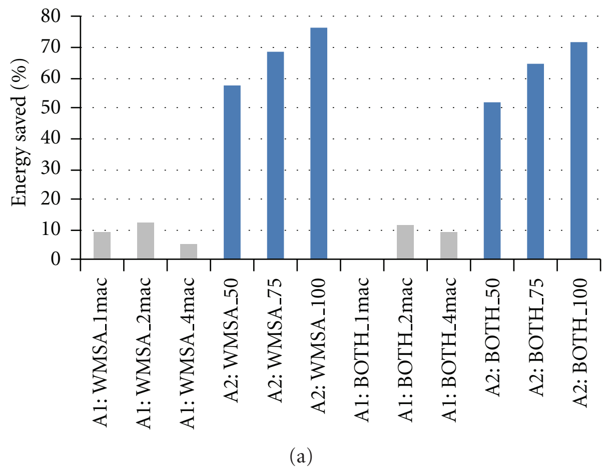


FIGURE 16: Energy savings in percent for A2, during adaptive switching at different operating frequencies (32-bit (a), 64-bit (b)).



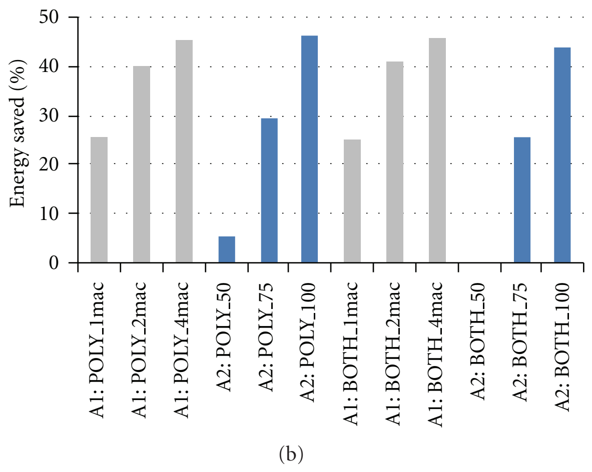


FIGURE 17: 32-bit: Energy savings in percent for A1 versus A2, normalized to worst case. WMSA (a), PI (b).

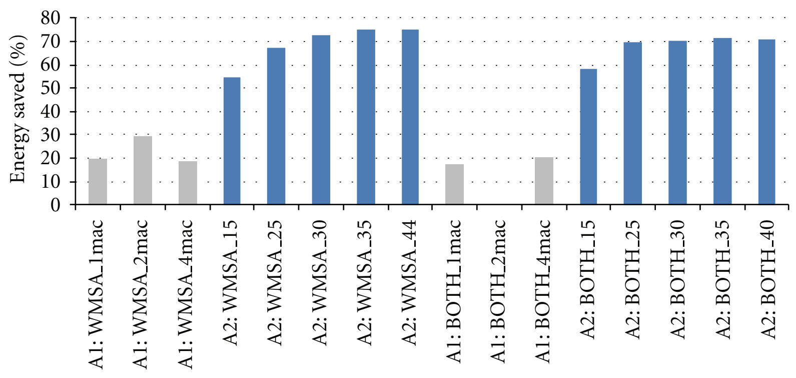


FIGURE 18: 64-bit WMSA: Energy savings in percent for A1 versus A2, normalized to worst case.

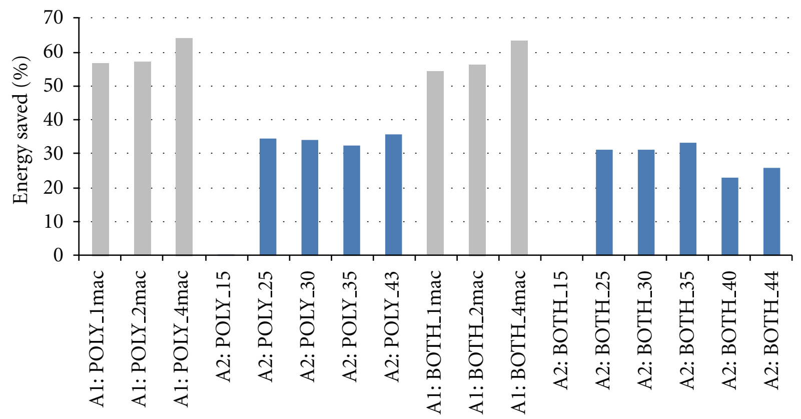


Figure 19: 64-bit PI: energy savings in percent for A1 versus A2, normalized to worst case.

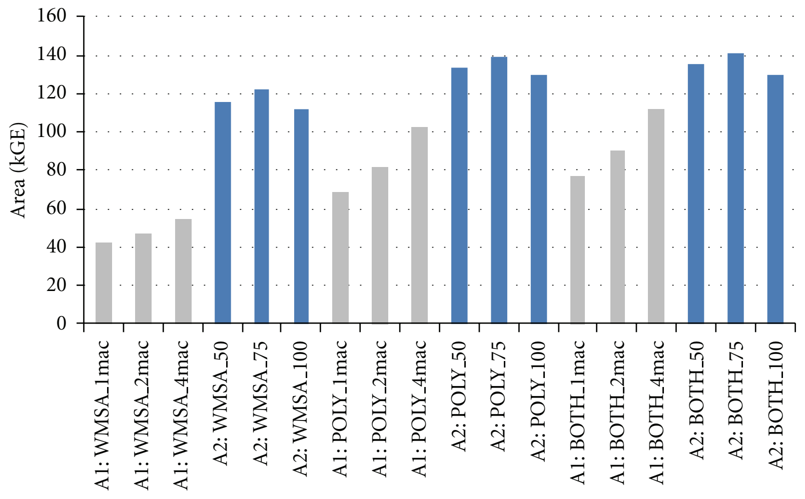


Figure 20: Area across all A1 and A2 design points for 32-bit designs.

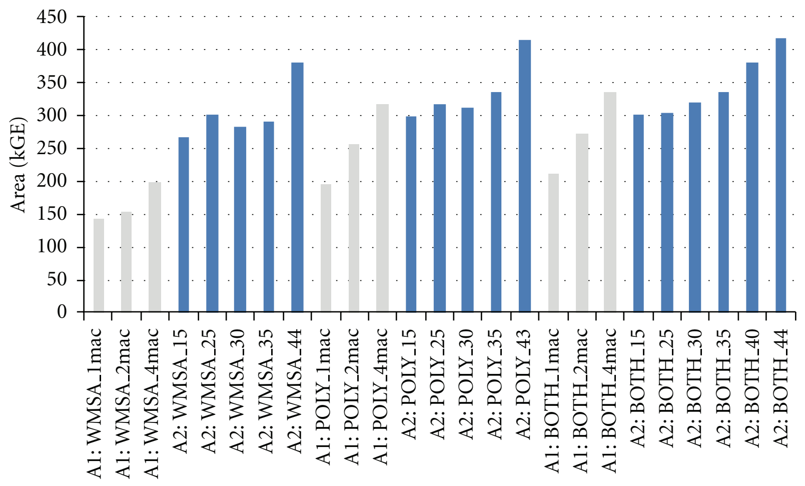


Figure 21: Area across all A1 and A2 design points for 64-bit designs.

coarse-grained core is forced to use interconnect structures of double bit width, incurring not only a greater area use, but also limiting operating frequency. Additionally, the bloated interconnect incurs extra power consumption, which cannot be neglected at higher bit widths.

The second observation to note is the large difference between the energy values for WMSA, A2 having a far better efficiency (Figures 17(a) and 18). This is due to having WMSA processing done on the smallest processing elements in A2, and not using the big MAC unit for plain additions, as is the case in A1.

Energy per symbol comparison of the two classes is as follows:

(i) for 32-bit WMSA (Figure 17(a)), A2 running at 100 MHz has much better energy values ($\geq 3\times$ *less*) than all design points of A1,

(ii) for 32-bit PI (Figure 17(b)), A2 performance is comparable with that of A1 ($\leq 3\%$ difference),

(iii) for 64-bit WMSA (Figure 18), A2 has much better energy (*up to* $4\times$ *less* than A1),

(iv) for 64-bit PI (Figure 19), A2 has much worse energy values (80% increase over A1). In this case, the power overhead of the large bit widths of the interconnect structure and the elements itself cannot be covered by an increase in throughput any more.

For small bit widths, A2 would be best for implementation, as the energy saves for the adaptive switching are much greater than in the case of A1. A1 must be chosen when predominantly bad signal conditions are expected and high precision hardware is needed.

Area evaluation is shown in Figures 20 and 21 and discussed as follows:

(i) due to area values spiking at higher frequencies, considering the energy values, the best frequency for A2 32-bits is 100 MHz and A2 64-bits is 35 MHz,

(ii) 32-bit, 64-bit WMSA, PI: A1 has much less area than A2 for dedicated structures,

(iii) 32-bit, 64-bit combined structures: area becomes comparable, with a difference of only 18.07 kGE for 32-bits and a smaller area (by 0.02 kGE) for 64-bits. At very low operating frequencies A2 area values are getting below the ones for A1 in case of PI, but energy efficiency is much worse than that of A1 at those points.

From the *flexibility* point of view, both A1 and A2 offer the necessary adaptiveness to accommodate the useful range of parameters of both algorithms. Both architectures can be directly tuned from assembly program level (configuring qualifiers, size of the sliding window, polynomial order, number of pilots, etc).

A2, however, due to its coarse grained core, has more flexibility and resources than those strictly needed for the two algorithms; hence, it can allow a more efficient execution by better tailoring the hardware to the application. Also, A2 can also be programmed to perform other external computations, on the idle elements during channel estimation processing. Given the fact that only a fraction of the hard deadline imposed by the WCDMA standard is needed to complete processing, it may be even possible to use the same structure for processing other blocks in the WCDMA receiver chain with some extensions to the load/store patterns from a new ASM chart for the new block. This would just create a new assembly program of the new block, which can be loaded after channel estimation processing is done, or a new hybrid one can be created.

Such flexibility can be a great advantage when aiming SDR blocks.

8. Conclusion and Outlook

In the context of software defined radio, an analysis on how flexibility can influence area and power consumption has been conducted, using two WCDMA channel estimation algorithms with sufficient performance and complexity difference. Within a scenario of saving energy when switching from a complex high-performance algorithm to an inferior less complex algorithm during operation in favorable channel conditions, we have shown that a higher degree of flexibility can yield significant energy savings (up to 97%); however, considerable savings (of up to 88%) can still be attained when carefully designing for the *exact* amount of flexibility required to support both algorithms. The evaluation across 25 design points and two architectural classes of different flexibility experimentally supports our findings.

Furthermore, we have proposed design guidelines to adapt and exploit a commercially available high-level synthesis tool, encouraging designers to explore flexibility of different architectural classes for SDR implementations.

For future work, we intend to apply the insights gained for designing blocks for newest LTE wireless standards. Design methodology and architectures using coarse-grained reconfigurable cores must be further explored, to open the door towards architectures which provide enough adaptability and flexibility to realize building blocks for a true SDR system.

Acknowledgment

The authors kindly acknowledge the support of Ultra High-Speed Mobile Information and Communication (UMIC) research cluster within the German Federal and State Government Excellence Initiative for funding this project.

References

[1] J. Mitola III, “Cognitive radio for flexible mobile multimedia communications,” in *Proceedings of the IEEE International Workshop on Mobile Multimedia Communications (MoMuC ’99)*, San Diego, Calif, USA, 1999.

[2] J. Mitola, “Cognitive radio architecture evolution,” *Proceedings of the IEEE*, vol. 97, no. 4, pp. 626–641, 2009.

[3] Y. Lin, H. Lee, M. Woh et al., “SODA: a low-power architecture for software radio,” in *Proceedings of ACM SIGARCH Computer Architecture News*, vol. 34, IEEE Computer Society, 2006.

[4] K. Van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss, "Vector processing as an enabler for software-defined radio in handheld devices," *Eurasip Journal on Applied Signal Processing*, vol. 2005, no. 16, pp. 2613–2625, 2005.

[5] M. Alles, T. Vogt, and N. Wehn, "FlexiChaP: a reconfigurable ASIP for convolutional, turbo, and LDPC code decoding," in *Proceedings of the 5th International Symposium on Turbo Codes and Related Topics (TURBOCODING '08)*, pp. 84–89, September 2008.

[6] B. Bougard, B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins, "A coarse-grained array accelerator for software-defined radio baseband processing," *IEEE Micro*, vol. 28, no. 4, pp. 41–50, 2008.

[7] J. Glossner, D. Iancu, M. Moudgill et al., "The sandbridge SB3011 platform," *Eurasip Journal on Embedded Systems*, vol. 2007, Article ID 56467, 2007.

[8] E. M. Witte, F. Borlenghi, G. Ascheid, R. Leupers, and H. Meyr, "A scalable VLSI architecture for soft-input soft-output single tree-search sphere decoding," *IEEE Transactions on Circuits and Systems II*, vol. 57, no. 9, pp. 706–710, 2010.

[9] T. Vogt and N. Wehn, "A reconfigurable ASIP for convolutional and turbo decoding in an SDR environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1309–1320, 2008.

[10] X. Chen, A. Minwegen, Y. Hassan et al., "FLEXDET: flexible, efficient multi-mode MIMO detection using reconfigurable ASIP," in *Proceedings of IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 69–76, IEEE, 2012.

[11] G. K. Rauwerda, P. M. Heysters, and G. J. M. Smit, "Towards software defined radios using coarse-grained reconfigurable hardware," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 3–12, 2008.

[12] V. Derudder, B. Bougard, A. Couvreur et al., "A 200 Mbps + 2.14 nJ/b digital baseband multi processor system-on-chip for SDRs," in *Proceedings of the Symposium on VLSI Circuits*, pp. 292–293, June 2009.

[13] H. Holma and A. Toskala, *WCDMA for UMTS*, vol. 4, Wiley Online Library, 2000.

[14] Calypto, http://calypto.com/en/products/catapult/overview.

[15] GAUT High Level Synthesis, http://www-labsticc.univ-ubs.fr/www-gaut/.

[16] Bluespec, http://www.bluespec.com/.

[17] Tensilica, http://www.tensilica.com/.

[18] Synopsys Designware ARC, http://www.arc.com/.

[19] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proceedings of the European Conference on Design and Test*, IEEE Computer Society, 1995.

[20] A. Chattopadhyay, H. Meyr, and R. Leupers, *LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation*, Morgan Kaufmann, 2008.

[21] Synopsys, http://www.synopsys.com/Systems/BlockDesign/processorDev.

[22] G. Yue, X. Zhou, and X. Wang, "Performance comparisons of channel estimation techniques in multipath fading CDMA," *IEEE Transactions on Wireless Communications*, vol. 3, no. 3, pp. 716–724, 2004.

[23] S. Abeta, M. Sawahashi, and F. Adachi, "Performance comparison between time-multiplexed pilot channel and parallel pilot channel for coherent Rake combining in DS-CDMA mobile radio," *IEICE Transactions on Communications*, vol. E81-B, no. 7, pp. 1417–1425, 1998.

[24] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," in *Proceedings of the International Conference on Computer Design: VLSI in Computer & Processors (ICCD '92)*, pp. 282–288, September 2002.

[25] S. Rajagopal, S. Rixner, and J. R. Cavallaro, "A programmable baseband processor design for software defined radios," in *Proceedings of the 45th Midwest Symposium on Circuits and Systems*, vol. 3, pp. III413–III416, August 2002.

[26] B. Mei, A. Lambrechts, D. Verkest, J. Y. Mignolet, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 90–101, 2005.

[27] C. Ebeling, D. Cronquist, and P. Franklin, "Rapid reconfigurable pipelined datapath," in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126–135, 1996.

[28] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[29] E. Waingold, M. Taylor, D. Srikrishna et al., "Computer baring it all to software: raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[30] K. Karuri, A. Chattopadhyay, X. Chen et al., "A design flow for architecture exploration and implementation of partially reconfigurable processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1281–1294, 2008.

[31] A. DeHon, "Density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[32] H. Fujisawa, M. Saito, S. Nishijima et al., "Flexible signal processing platform chip for software defined radio with 103 GOPS dynamic reconfigurable logic cores," in *Proceedings of IEEE Asian Solid-State Circuits Conference (ASSCC '06)*, pp. 67–70, November 2006.

[33] O. Gustafsson, K. Amiri, D. Andersson et al., "Architectures for cognitive radio testbeds and demonstrators—an overview," in *Proceedings of the 5th International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCom '10)*, pp. 1–6, June 2010.

[34] P. D. Sutton, J. Lotze, H. Lahlou et al., "Iris: an architecture for cognitive radio networking testbeds," *IEEE Communications Magazine*, vol. 48, no. 9, pp. 114–122, 2010.

[35] S. Rajagopal, S. Bhashyam, J. R. Cavallaro, and B. Aazhang, "Real-time algorithms and architectures for multiuser channel estimation and detection in wireless base-station receivers," *IEEE Transactions on Wireless Communications*, vol. 1, no. 3, pp. 468–479, 2002.

[36] S. Rajagopal, B. Jones, J. Cavallaro et al., "Task partitioning wireless base-station receiver algorithms on multiple dsps and fpgas," in *Proceedings of the International Conference on Signal Processing, Applications, and Technology (ICSPAT '00)*, Dallas, Tex, USA, 2000.

[37] S. Rajagopal, S. Bhashyam, J. R. Cavallaro, and B. Aazhang, "Efficient VLSI architectures for multiuser channel estimation in wireless base-station receivers," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 31, no. 2, pp. 143–156, 2002.

[38] A. Kuehlmann and R. A. Bergamaschi, "High-level state machine specification and synthesis," in *Proceedings of the International Conference on Computer Design: VLSI in Computer & Processors (ICCD '92)*, pp. 536–539, 1992.

[39] J. David and E. Bergeron, "An intermediate level HDL for system level design," in *Proceedings of the Forum on specification & Design Languages (FDL '04)*, pp. 526–536, 2004.

[40] Z. Wang, Z. Rakosi, X. Wang, and A. Chattopadhyay, "ASIC synthesis using architecture description language," in *Proceedings of the International Symposium on VLSI Design, Automation and Test (VLSI-DAT '12)*, 2012.