Customizing VLIW processors from dynamically profiled execution traces

Gorker Alp Malazgirt, Arda Yurdakul, Smail Niar

Microprocessors and Microsystems 39 (2015) 656–673

Customizing VLIW processors from dynamically profiled execution traces

Gorker Alp Malazgirt [a,*], Arda Yurdakul [a], Smail Niar [b]

[a] Bogazici University, Computer Engineering Department, Istanbul, Turkey
[b] LAMIH – University of Valenciennes, Valenciennes, France

Keywords:

VLIW
Dynamic profiling
Genetic algorithm
Mathematical model
FPGA
ASIC

abstract

The design philosophy of VLIW processors is to maximize instruction level parallelism (ILP) starting from compiler and machine code level to all the way down to memory and computational blocks. For this purpose, VLIW tailoring has been an important research area, because non-tailored VLIWs cannot fully utilize the available VLIW hardware resources. This paper introduces a method which achieves VLIW customization by processing execution traces obtained by dynamic profiling. Our method differentiates memory and non-memory instructions while processing execution traces. Customizing VLIW multi-port memory from memory operations provides better memory utilization and higher performance. Moreover, exploration of the multi-port memory configuration is coupled with data path exploration, namely the number and the composition of execution units for efficient extraction of ILP. We have designed a genetic algorithm for the exploration of the large design space formed by the execution traces. Our experiments show that our method has improved and found more compact memory topologies than state-of-the-art VLIW customization algorithms. In addition, we compare the execution performance, power consumption, average parallelism and area-delay product results of our VLIW model with a RISC processor model on evaluated benchmarks using our simulator framework.

1. Introduction and motivation

Embedded computing with application specific VLIW (Very Long Instruction Word) based processors has long been an alternative to superscalar processors to run applications in many areas such as digital image processing, telecommunications and consumer electronics. VLIW's long instruction words encode the concurrent operations, which are decided at compile time. This explicit encoding leads to reduced hardware complexity compared to a high degree superscalar out-of-order implementations of Reduced Instruction Set Computers (RISCs) or Complex Instruction Set Computers (CISCs), because unlike superscalar out-of-order processors, the VLIW hardware is not responsible for discovering instruction level parallelism (ILP). VLIW architectures, however, require more compiler support than RISC and CISC [1,2].

Although VLIW hardware is simpler than superscalar out-of-order RISC and CISC, they must be designed efficiently because executing multiple operations concurrently comes with the cost of having wider instruction memory, data memory and more complex interconnections. Data memory needs higher number of ports for providing multiple data to different execution units. Thus, the area of the data memory increases [3]. Larger memory with higher memory bandwidth requirements and execution units must be connected with larger interconnection networks which increase total area and decrease performance.

Tailoring a VLIW processor for a specific application necessitates the extraction of concurrent operations. Then, concurrent operations are bundled as instructions exploiting the available instruction level parallelism. Numerous studies have shown that VLIW customization using ILP information can generate performance improvements [2,4–6].

ILP information has been either statically or dynamically produced. With static profiling, this information is collected by analyzing the program without executing it. In order to measure certain characteristics of a program, static profilers apply algorithms. Information like cache misses [7] is hard to extract with static profiling. Therefore, outputs of these static profiles are estimates in many cases. In addition, static profiling at basic block level can prevent extracting existing ILP because of creating poor schedules [33].

DOI : 10.1016/j.micpro.2015.09.005

Dynamic profiling allows program information to be collected during the execution of the program. A profile can contain information from several different executions of a program. Dynamic profiling is more accurate than static profiling, since it does not rely on estimates such as memory pointer aliasing but accurately captures the data flow information in the memory when the program is executed. Hence, all the irregular patterns can be captured by the profiler. Dynamic profiles are dependent on the input data set. Dynamic profiling has already been used in high level synthesis domain to generate better solutions than static approaches [8,9]. Similarly, our work also shows that dynamic profiling has generated better solutions than static profiling in application specific VLIW processor tailoring.

Multi-port memories are vastly used in VLIW processors and they are one of the most resource consuming on-chip modules [10,3]. Although VLIW processors allow more parallelism, they are difficult to utilize because of more memory ports and bandwidth requirements than RISC and CISC processors [11]. This is due to increasing code size and register usage for supporting more ILP. An under-utilized multi-port memory decreases data access times drastically and increases power consumption [3,12]. Due to these challenges, significant number of previous research has been focused on customizing the data path of VLIW processors with small and fixed number of ports in their multi-port memories [13,14]. Hence, the effort has been on ILP extraction for customization of the VLIW data path, namely the number of execution units. In contrast, our method extracts ILP to customize both the number of memory ports and the data path.

In this way, we achieve the highest performance with minimum memory size and highest memory utilization. Similarly, we can also extract the number, the composition and the connection scheme of execution units in the VLIW data path as shown in Fig. 1. Our method starts with extracting the port number of multi-port memory for minimum execution time, then uses this information to customize the number and composition of execution units. We aim to decrease the number of memory access steps by combining concurrent memory instructions. Although the number of memory instructions are constant, the reduction in memory access steps increase execution performance.

Our paper proposes a method which achieves VLIW customization by using instructions obtained from dynamic profiles. We generate:

- Minimum size multi-port memory that provides enough memory bandwidth for achieving the minimum execution time
- Customization of VLIW data path according to the selected multi-port memory
- Reducing number of memory access operations by allowing concurrent accesses to multi-port memory in a single VLIW instruction

Our method is designed to be used at the initial stage of the VLIW processor design. At the beginning, the application is profiled and the execution trace is extracted. At this stage, the number of the memory ports or the execution units are not known. Then, our tailoring rules described in Section 3.2 are applied on memory instructions for finding out the number of memory ports of the VLIW's multi-port memory. Based on the multi-port memory decision, the same tailoring methods are applied on the non-memory operations. After the number of ports of memory and the composition of execution units are found, a compiler with parametric backend capabilities can be used to compile the application [1,15].

Our paper has the following differences from state of the art methods. We differentiate memory instructions and non-memory operations while processing execution traces. We extract ILP from memory operations and use this parallelism information to extract the memory constrained ILP from the non-memory operations, i.e. the data path. We customize the number of memory ports based on memory operations. This information is used for customization of execution units. In contrast, state of the art either neglect memory operations or do not differentiate them from non-memory operations. This causes lower performance and memory utilization. Moreover, previous research that explores the number of memory ports is not coupled with data path exploration, namely the number and composition of execution units. Instead, the number of execution units are either fixed or they are template-based.

Our customization method aims to ensure that maximal performance is attained with the fewest number of memory ports and execution units. In addition, decreasing the number of execution units and identifying the composition of these units set forth an efficient approach as opposed to incorporation of homogeneous execution units on the data path. When number of memory ports increases, number of FUs has to increase in order to utilize the available ports and vice versa. Tailoring the composition of EUs

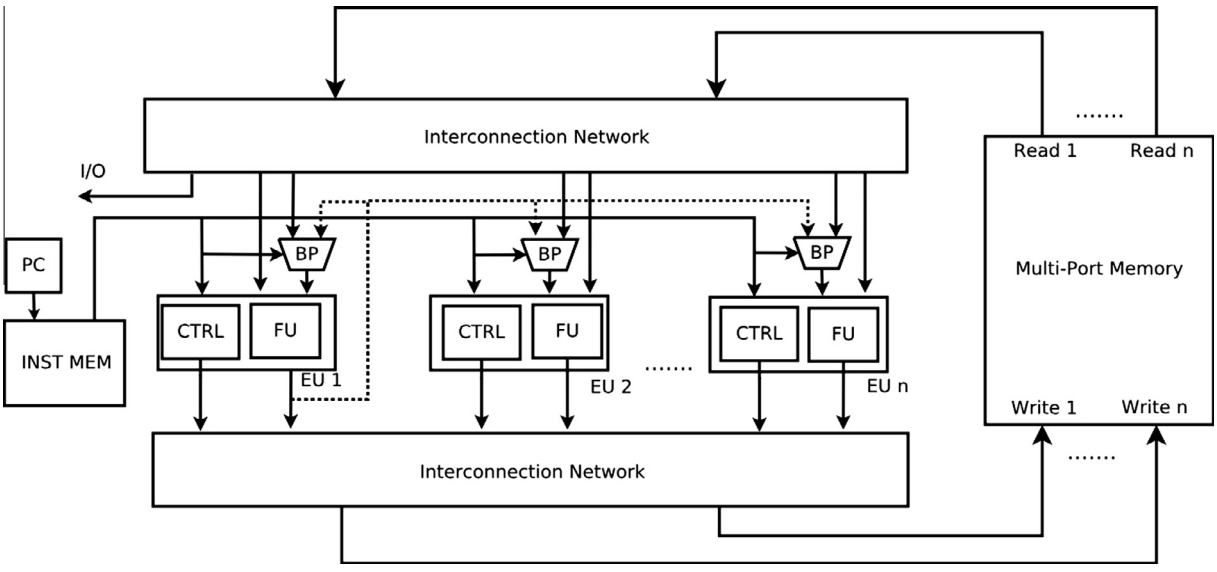

Fig. 1. VLIW architecture with functional units and multi-port memory.

DOI : 10.1016/j.micpro.2015.09.005

with respect to the application prevents generation of redundant hardware and maximizes resource utilization.

In the experimental section, we introduce numerous results. First of all, we present that our method can be used in different technologies, namely FPGA and ASIC. Second, we show that our method is able to customize algorithms from different domains such as compute and data intensive. We have compared our customization method with a recent method [16]. We show that our method recommends VLIW architectures with fewer number of ports and execution units than the state of art in order to provide same execution performance. In addition, we simulate the benchmarks and the compare execution performance and power consumption of benchmarks between our VLIW model and the RISC processor model which has been used extensively in FPGA and ASIC designs. We also present the area-delay product of data-memory results and show that our customized VLIWs can exploit the available ILP efficiently.

The rest of this paper is organized as follows. Next section presents our reference VLIW processor model and its building blocks. Section 3 details our tool flow and the customization methods are explained. Experimental results are presented in Section 4. We discuss the related works in Section 5 and the last section includes our discussion and conclusion.

2. Reference VLIW architecture model

The reference VLIW architecture is shown in Fig. 1. It is executes multiple independent instructions in each execution unit. The customizable parts of the VLIW are the number of execution units, the composition of functional units, the number of memory ports and necessary bypass blocks.

2.1. Execution unit

Each Execution Unit consists of a Functional Unit (FU) and a Control Unit (CTRL). Execution units are connected to multi-port memories via input and output interconnection network. Though, each EU can have different data widths, in this work, we require that each execution unit has two 32-bit inputs and one 32-bit output. Data forwarding is possible through bypass blocks (BP).

2.1.1. Functional and bypass blocks

Each FU as shown in Fig. 2 can contain an ALU for integer, floating point and logic operations. It also contains a Load/Store Unit which manages all load and store operations. Similarly, the branch unit (BR) handles the branch operations. The branch units are assumed to be multi-way branches with conditional execution capabilities and this design decision allows reordering of branch instructions [17]. All the blocks of the FU are controlled by each CTRL unit inside an EU. Only one block can be active inside an FU for each concurrent operation. When a VLIW instruction is decoded by the CTRL unit, the decoded instructions are executed by FUs. The blocks inside an FU are decided after the Maximum ILP optimization algorithm which is explained in Section 3.3.2. Therefore, if an application does not have any branch instructions, FUs are built without the BR units. On the other hand, if Maximum ILP optimization algorithm finds a reordering which facilitates concurrent executions of data independent branch instructions, then several FUs can employ the BR units. Bypass blocks allow output data of a FU to be forwarded as input to itself or another FU by replacing two consecutive load/store instructions with bypass instructions and bypass blocks. Thus, the output of one FU is written to bypass block registers instead of memory and the memory instruction is replaced with bypass instruction. Unlike the register file, bypass blocks are not general purpose memory. Instead, they are placed between the outputs and inputs of FUs and connected with multiplexers. Moreover, similar to memory ports, redundant bypass blocks should always be eliminated.

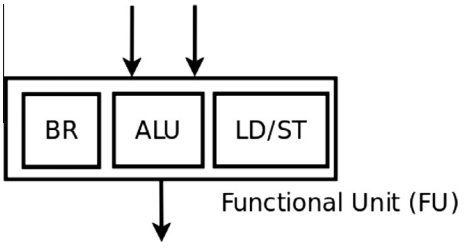

Fig. 2. Blocks of functional unit.

2.1.2. Control unit

Control Unit (CTRL) decodes the long instruction word and generates the signals to enable the blocks of FU. Therefore, CTRL logic must exist in each execution unit. CTRL is also responsible for enabling/disabling bypass blocks for data forwarding. Instruction memory provides RISC like instructions which are fixed width. Nevertheless the instructions are longer than RISC instructions in order to specify independent operations. Instruction structure is shown in Fig. 3. Each VLIW instruction word array encodes several operations. The width of the VLIW instruction depends on number of execution units. In the sub instruction, the valid bit allows predication, in other words, it indicates whether the instruction should be ignored. Each sub instruction encodes three operand addresses that are two read and one write. The available memory space is mapped to banks, therefore the compiler can schedule concurrent operations using available memory space in different banks. Opcode field selects which operation to execute and Function field selects a variant of the operation and enables/disables bypass block.

2.1.3. Interconnection network

We do not detail the implementation of the interconnection network in this paper. However, we assume that every execution unit can access memory through any port for reading and writing. This helps programmability and reduces compiler workload.

2.2. Memory architecture

The reference architecture uses the multi-port memory design in [18]. Each execution unit has an address space and this address space corresponds to a bank with different height and width. Each bank consists of replicated BRAMs in the FPGA implementation. Each memory bank is associated with a write port. In the FPGA implementation, inside a bank, all BRAMs hold the same data and represent the same address space range to increase the number of read ports. Banks have their own local address space and union of all banks forms the global address space so the global address space is the sum of all local address spaces of processing elements. This multi-port memory structure simplifies the work required by the reordering algorithm. During the write operation, it guarantees updating all the copies that is interfaced to the execution unit. In addition, both the Maximum ILP Extraction Algorithm explained in Section 3.2 and the Maximum ILP Optimization Algorithm explained in Section 3.3.2 prevent data conflicts. When these algorithms are executed, the data which are used by the sub-instructions of a VLIW instruction are arranged such a way that each sub-instruction accesses a different memory bank in the memory architecture at each access step. This guarantees prevention of data conflicts in a VLIW instruction.

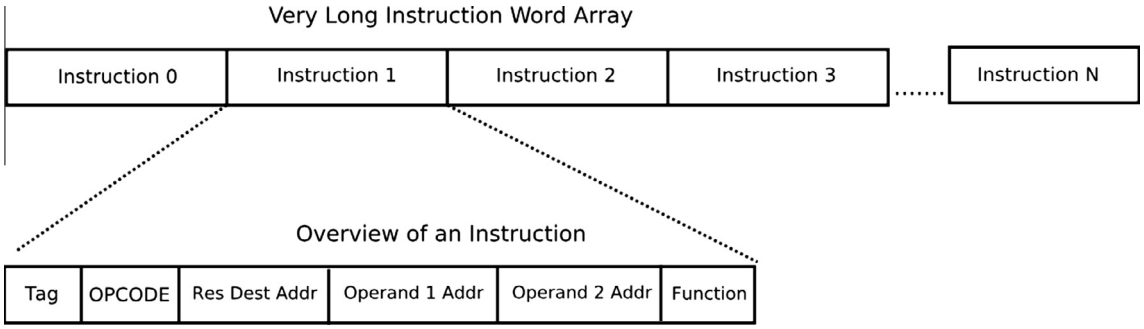

Fig. 3. VLIW instruction word array.

3. Tailoring of VLIW architectures

In this section, we detail the steps of our VLIW customization method. We start explaining the dynamic profiling. We further detail our tool flow with a sample example. Then our maximum ILP extraction method is explained. Before we discuss the details of our evolutionary algorithm, we explain the mathematical model.

3.1. Dynamic profiling of applications

For application specific dynamic profiling, we developed a software tool which can use any binary instrumentation library to extract executed instruction traces. After capturing traces, the tool creates separate memory and non-memory trace files. Our VLIW customization algorithms work on the instructions provided by the trace. Therefore, our tool provides the flexibility of processing well-known ISAs as well as custom instruction traces provided by the designer. Since code generation is the responsibility of the compiler, our profiling tool does not take part in the code generation.

A sample memory load/store instruction trace file is shown in Fig. 4(a). It contains only load/store instructions, thus non-memory instructions are filtered. In figure, *Access Step* column represents the global order of the memory instruction. This order contains memory and non-memory instructions, it helps to preserve data dependencies. *Instruction Address* column stores the virtual addresses obtained from our R2 profiling tool. *Instruction Signature* column makes us differ two identical memory access instructions that take place at different access steps. For instance in access steps "1" and "23", there are two mov instructions at address *M1*. However, the second mov instruction *M1* at access step 23 is 22 access steps ahead of the first one. So we use two different signatures to distinguish these instructions. Signatures help us to record parallelism across the loop boundaries. Memory load/store instructions have source and destination arguments, therefore *Arg 1*, *Arg 2* and *R*/*W* columns respectively identify source, destination and Load/Store type of instruction at given access step.

Using memory access trace, data dependency table is generated. This is done by analyzing true register dependencies and memory read/write orders in the trace. While traversing through the trace we bind instructions with their memory/register usage. This binding is kept in a table. Our parallelism analysis and memory reduction algorithms query for lifetimes of registers and memory addresses for given instructions at each access step. In addition, *Data Address* column stores the actual memory address that is inferred by *Arg 1* or *Arg 2* columns.

Data dependency analysis using data addresses is equivalent of pointer alias analysis [19] with perfect accuracy, because, after dynamic profiling, actual memory address referred by a load or store can be processed. A dependency or a conflict can be extracted when different instructions access the same address location. Similarly, branch and indirect jump predictions are redundant when instruction traces convey all of these information. Instead, the processing time is spent for ILP extraction and instruction rescheduling for resource optimization such as the number of memory ports.

The non-memory instruction traces must also be processed in order to fully exploit the parallelism from a given execution trace. The number and composition of VLIW execution units are decided after exploring the parallelism from the non-memory instruction traces. Thus, we apply the previously explained procedures to non-memory instructions. As shown in Fig. 4, *Access Step* and *Instruction Signature* are used for global ordering of instructions and identifying identical instructions. *Arg #* represent data locations which can be a register, a memory address. For data analysis, data dependency table is also generated. Both Maximum ILP Extraction Algorithm explained in Section 3.2 and the Maximum ILP Optimization Algorithm explained in Section 3.3.2 can process non-memory and memory instruction traces.

3.2. Exploring maximum ILP from instruction traces

In order to extract the maximum and average parallelism from instruction traces, we have designed a single pass algorithm that schedules instructions to access steps and guarantees data dependency. Maximum parallelism is important because it yields the maximum amount of memory access or FU usage in an instruction bundle. Similarly, average parallelism gives an idea of how much ILP is extracted from a given application.

We define maximum parallelism as

$$p_{max} = \max_{t}(n_t) \tag{1}$$

where n_t is the number of memory access instructions at access step t. The parallelism exploration schedules and bundles instructions in the trace file without breaking data dependencies. Hence, we propose the rescheduling logic rules with the following definitions.

Definition 1 (*Instruction Signature*). Within the trace or a portion of the trace, let there be m instructions $\mathbb{I} = \{I_1, I_2, \ldots, I_m\}$ with execution times $\{e_1, e_2, \ldots, e_m\}$. We define the instruction signature as $S_{i,k} = \langle I_i, A_x, \rho, k, e\rangle \in \mathbb{S}$ where

- $\mathbb{S}$ is the set of all instruction signatures.
- A_x is data memory address and range of x is as big as the data memory/register file size.
- ρ denotes memory access operations, i.e., $\rho \in \{read, write\}$.
- k represents the kth occurrence of an instruction.
- e represents the execution time of the signature.

Definition 2 (*Rescheduling Logic Rules*). Let S_i, S_j be any two signatures in $\mathbb{S}$. For all $k > 0$, all signatures can be moved to an earlier access step provided that the following rules are not violated:

Access Step	Instruction Address	Instruction Signature	Instruction Mnemonic	Arg #1	Arg #2	R/W	Data Address
1	M1	S1,1	mov	[r5-0x50]	r2	W	D5120
2	M2	S2,1	mov	r1	[r5-0x50]	R	D5120
3	M3	S3,1	cmp	r1	[r5-0x54]	R	D5660
4	M4	S5,1	mov	r1	[r5-0x50]	R	D5120
5	M5	S6,1	cmp	r7	[r5-0x70]	R	D5520
12	M6	S8,1	mov	[r5-0x50]	r1	W	D5120
13	M7	S11,1	mov	r8	[r5-0x28]	R	D5060
14	M8	S12,1	mov	r9	[r6+r4*1]	R	D6960
15	M9	S15,1	mov	r10	[r5+r4*4-0x40]	R	D5150
16	M10	S16,1	mov	[r5-0x44]	r1	W	D5540
17	M11	S17,1	mov	r1	[r5-0x44]	R	D5540
18	M12	S19,1	mov	r3	[r5-0x50]	R	D5120
19	M13	S20,1	mov	r6	[r5-0x28]	R	D5060
20	M14	S21,1	mov	r3	[r6+r4*1]	R	D6970
21	M15	S24,1	and	r4	[r5+r4*4-0x40]	R	D5170
22	M16	S25,1	mov	[r5-0x44]	r1	W	D5540
23	M1	S1,2	mov	[r5-0x50]	r2	W	D5120
24	M2	S2,2	mov	r1	[r5-0x50]	R	D5120

(a)

Access Step	Instruction #1	Instruction #2	Instruction #3	Instruction #4	Instruction #5
1	S1,1/D5120/R				
2	S2,1/D5120/W	S3,1/D5660/R			
3	S5,1/D5120/R	S6,1/D5520/R			
4	S8,1/D5120/W	S11,1/D5060/R	S12,1/D6960/R	S15,1/D5150/R	S16,1/D5540/W
5	S17,1/D5540/R	S19,1/D5120/R	S20,1/D5060/R	S21,1/6970/R	S24,1/5170/R
6	S25,1/D5540/W	S1,2/D5120/R	S2,2/D5120/W		

(b)

Access Step	Instruction #1	Instruction #2	Instruction #3
1	S1,1	S11,1	S12,1
2	S2,1/ Rule 3	S3,1	S20,1
3	S5,1/ Rule 2	S6,1	S24,1
4	S8,1/ Rule 3	S16,1	S15,1
5	S19,1/ Rule 2	S17,1/ Rule 2	S21,1
6	S25,1/ Rule 3	S1,2	S2,2

(c)

Fig. 4. Memory port size optimization example by using memory instruction traces. (a) Extracted memory trace file from a sample sequential code. (b) Maximum parallelism obtained after Maximum ILP Extraction Algorithm is executed. Issue slot size = 5, Memory ports = 5R 2W. (c) Optimized parallelism with minimum memory port size after executing Maximum ILP optimization heuristic. Issue slot size = 3, Memory ports = 3R 1W.

- Rule 1: $\langle S_i, A_x, write, k+1\rangle$ cannot start before or at the same time with $\langle S_j, A_x, write, k\rangle$
- Rule 2: $\langle S_i, A_x, read, k+1\rangle$ cannot start before or at the same time with $\langle S_j, A_x, write, k\rangle$
- Rule 3: $\langle S_i, A_x, write, k+1\rangle$ cannot start before or at the same time with $\langle S_j, A_x, read, k\rangle$

The instructions satisfying these rules exploit ILP without breaking data dependencies. All rules are concerned with data hazards that may happen when write and read to same address occur by different instructions. Rule 3 also handles register data dependency. All the rules are checked in a single pass of memory instructions. Figs. 5 and 4(c) show the data dependency obtained after applying the rescheduling logic rules to Fig. 4(a). In the figure, edges represent data dependencies between instructions. Nodes that have no in- or out-edges can be scheduled independently at any available access step.

Algorithm 1 shows the steps of Maximum ILP Extraction Algorithm. The input of the algorithm is an instruction trace file similar to Fig. 4(a) and the data dependency graph. The algorithm iteratively applies rescheduling logic rules and bundles signatures to access steps. Thus, the original dependencies are preserved. The output of the algorithm is a new instruction trace file as shown in Fig. 4(b). We call the new instruction trace file as rescheduled instruction trace file. In the algorithm, for each signature, rescheduling logic rules and data dependencies are checked. Signatures that satisfy all the rescheduling rules and data dependencies are scheduled at the same access step. Otherwise, signatures are scheduled to the next access steps.

Maximum ILP Extraction Algorithm can be both applied to memory and non-memory signatures. The Rescheduling Logic Rules do not break the original order of instructions. We first apply the algorithm to memory signatures. Then, algorithm is applied to non-memory signatures. The maximum parallelism can be easily calculated by using Eqs. (2) and (3). For memory instructions, the number of memory read write ports can be calculated as follows:

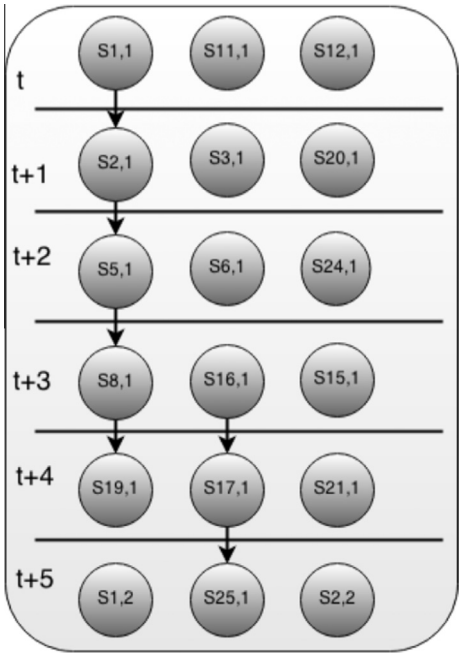

Fig. 5. Data dependencies between memory instructions are shown with edges between vertices, vertices can also have dependencies with registers instead of memory address.

$$MEM_{write} = \max_t(w_t) \tag{2}$$

$$MEM_{read} = \max_t(r_t) \tag{3}$$

where w_t and r_t is the number of write and read instructions, respectively, at access step t.

Let λ be defined as the number of access steps in the rescheduled instruction trace file. Then average parallelism can also be calculated as follows:

$$p_{ave} = \left\lceil \frac{\sum_{t=1}^{\lambda} n_t}{\lambda} \right\rceil \tag{4}$$

Algorithm 1. Maximum ILP Extraction Algorithm

Algorithm 1. Maximum ILP Extraction Algorithm

Input: Instruction Trace File and Data Dependency Graph
Output: Rescheduled Instruction Trace File
1 Rescheduled Instruction Trace File, is initialized as empty;
2 *Rescheduled_Access_Step_Pointer*, which points to the already scheduled instructions in Rescheduled Instruction Trace File, is initialized with 1;
3 **foreach** signature in the Instruction Trace File **do**
4 Check data dependencies and apply Rescheduling Logic Rules to the signature against the signatures pointed by *Rescheduled_Access_Step_Pointer*;
5: **if** all the reordering rules are satisfied and no data dependencies exist **then**
6 add signature to the step pointed by *Rescheduled_Access_Step_Pointer* in Rescheduled Instruction Trace File;
7 **else**
8 increment *Rescheduled_Access_Step_Pointer*;
9 add signature to the new access step;
10 **end**
11 **end**

3.3. Optimizing resources for VLIW tailoring

Maximum ILP extracted by Algorithm 1 is important because fastest execution requires maximum parallelism. However, implementations based on maximum parallelism will be inefficient in terms of area and power consumption. Therefore, we have designed a heuristic to reduce maximum parallelism. We optimize VLIW instruction bundles by reordering instructions to different access steps. This helps to gain significantly from resource usage while keeping the performance attained at maximum parallelism.

We rely on the mathematical model while developing our heuristic. In the following subsections, we firstly present our mathematical model, then explain the related heuristic.

3.3.1. Mathematical model

We assume that data dependency is stored as a graph. This graph is a polar directed non-hierarchical acyclic graph $G_s(V,E)$ where the vertex set $V = \{v_i; i = 0,1,\ldots,m\}$ is in one-to-one correspondence with the set signatures and the edge set $E = \{(v_i, v_j); i,j = 0,1,\ldots,m\}$ represents dependencies between operations. The formal model of resource optimization can be achieved by using binary decision variables with two indices: $W = \{w_{i,t}; i = 0,1,\ldots,m; t = 1,\ldots,\lambda\}$ and $R = \{r_{i,t}; i = 0,1,\ldots, m; t = 1,\ldots,\lambda\}$. The set of W consists of instructions which write to memory and the set of R consists of operations which read from memory or a register. The indices of the binary variables relate to

the instructions and access steps. Thus, a binary variable, $w_{i,t}$, is 1 only when signature v_i is scheduled at access step t and writes to memory or a register. λ represents the number of access steps calculated by Algorithm 1. The first three constraints are formal models of our rescheduling rules presented in the previous section. In other words Rule 1 guarantees write-after-write, Rule 2 guarantees write-after-read and Rule 3 guarantees read-after-write sequencing.

- Constraint 1 = Rule 1:

$$\sum_{t=1}^{\lambda} t \cdot w_{i,t} - \sum_{t=1}^{\lambda} t \cdot w_{j,t} \geqslant 1 \quad \forall i,j = 1,\ldots,m : (v_i, v_j) \in E \tag{5}$$

- Constraint 2 = Rule 2

$$\sum_{t=1}^{\lambda} t \cdot r_{i,t} - \sum_{t=1}^{\lambda} t \cdot w_{j,t} \geqslant 1 \quad \forall i,j = 1,\ldots,m : (v_i, v_j) \in E \tag{6}$$

- Constraint 3 = Rule 3

$$\sum_{t=1}^{\lambda} t \cdot w_{i,t} - \sum_{t=1}^{\lambda} t \cdot r_{j,t} \geqslant 1 \quad \forall i,j = 1,\ldots,m : (v_i, v_j) \in E \tag{7}$$

We also have to satisfy that each instruction should not appear more than once in any access step. Hence, the next two constraints are formulated to realize this situation:

- Constraint 4

$$\sum_{t=1}^{\lambda} w_{i,t} = 1 \quad \forall i = 1,\ldots,m \tag{8}$$

- Constraint 5

$$\sum_{t=1}^{\lambda} r_{i,t} = 1 \quad \forall i = 1,\ldots,m \tag{9}$$

Each binary decision variable which is bound to an instruction must exist in exactly one access step.

Our final constraint formulates p_{max} given by Eq. (1). The model tries to minimize the number of instructions at any access step t without changing the access steps:

- Constraint 6

$$p_{max} \geqslant \sum_{t=1}^{\lambda} w_{i,t} + \sum_{t=1}^{\lambda} r_{j,t} \quad i,j = 1,\ldots,m \tag{10}$$

The objective is to minimize the maximum number of instructions at all access steps, so as to lower the number of memory ports and number of concurrent operations.

$$\min p_{max} \tag{11}$$

Decision variables are summarized in Table 1. The formal model has provided insights and forms the foundation of the genetic algorithm (GA) which is explained in next section. Moreover, additional constraints and problem extensions could easily be incorporated into the model for further improvements.

3.3.2. Maximum ILP optimization heuristic

The genetic algorithm (GA) is designed to increase the utilization of VLIW without decreasing the maximum performance found by Algorithm 1. This is achieved by finding new schedules with more uniform instruction bundles at each access step. The algorithm is based on a genetic algorithm (GA) in the literature [20]. The GA has been used in many domains but it requires fine tuning such as population size and termination conditions. There does not exit a single GA configuration that works for all problems. We have made several experiments and selected the GA parameters in order to reduce exploration time and obtain good solutions. We have selected GA because the instruction encoding could be converted to a chromosome encoding in a very simple way. In addition, GA allows to solve multi solution problems because of its population concept. The population allows designers to process the solutions which are favored by the GA at the time of the execution. We only implement the existing best solution in the population when GA.

Table 1
Decision variables used in the model.

Decision variable	Definition
$r_{i,t}$	1 if read operation i is executed at access step t, 0 otherwise
$w_{j,t}$	1 if write operation j is executed at access step t, 0 otherwise
$p_{max,\alpha}$	Maximum parallelism within a window of α access steps

The GA applies global scheduling, however the distance of rescheduling instructions from their original access step locations are bounded. We call this distance the evaluation window. α represents the evaluation window size. An instance of GA is initialized and run in an evaluation window until its termination criteria is met. This allows us to decrease the design space and apply GA successfully.

The GA processes the rescheduled trace file generated by Algorithm 1 and produces a new trace file. For example, the trace file in Fig. 4(b) is input to GA and the trace file showing the final schedule in Fig. 4(c) is produced. As observed from the figures, our heuristic has managed to decrease maximum parallelism (p_{max}) to 3 from 5, and as expected, the number of read and write ports has also decreased.

In Fig. 4(c), we also present which Rescheduling Logic Rules are applied to the signatures by adding applied rule next to the signature. For example: Rule 1 is applied to Signature S2,1. The rules are only applied to the instructions which have data dependency. Therefore, the signatures that do not have any rules indicate that Rescheduling Logic Rules haven't been applied to them.

In addition, the GA identifies memory read signatures and remove them if they satisfy the signature removal rule explained below.

Definition 3 (*Signature Removal Rule*). Let S_i, S_j be any two data read signatures in $\mathbb{S}$. For all $k > 0$, one of the duplicate read signatures is removed if following condition is satisfied:

- Rule 4: $\langle S_j, A_x, read, k\rangle$ is scheduled at the same access step $\langle S_i, A_x, read, k\rangle$

The signature removal rule reduces the number of read instructions in an instruction bundle by removing duplicate read instructions scheduled at the same access step during the execution of GA. When duplicate read instructions are removed, it allows GA to schedule other instructions to these available slots. As a result, total number of memory instructions is reduced. In addition, removing duplicate read instructions in an instruction bundle may also reduce the number of memory ports when these slots are kept empty.

In an evaluation window, instructions are selected for rescheduling from access steps that have higher number of instructions than p_{ave}. These selected instructions are rescheduled to access steps that have lower number of instructions than p_{ave}. Evaluation windows traverse trace files linearly and there is no overlapping between evaluation windows. In the next evaluation window, a new instance of GA is initialized and this process

repeats from the beginning of the instruction trace file until the end.

For each evaluation window, the value of α is chosen from profiling information. Our dynamic profiler tool provides the upper bound on α based on checking inter-iteration loop and data dependencies or indirect jump blocks. As an example, in Fig. 4(b), α is 6. This is determined by the profiler; the loop bound is 22 instructions which are reordered to 6 access steps in exploring maximum ILP step.

The GA can be applied to memory and non-memory instructions. Yet, we first start by optimizing memory instructions and find the number of read/write memory ports. Then, algorithm is applied to non-memory instructions. We prevent the number of execution units to exceed the number of memory ports. When GA is applied on non-memory instructions, the number of FUs and the composition of FUs are identified. Following sections detail the important parameters of the GA.

Solution encoding: We have observed that chromosome encoding can represent VLIW instruction bundles without much effort. Each encoding is designed as one dimensional array. Each element of the array represents an instruction. This array is called the chromosome and each instruction is a gene of the chromosome. The example shown in Fig. 5 is encoded as shown in Table 2. There are eighteen instructions in the given example. Each array element holds a value. This value is the access step each instruction is scheduled. For example, instructions $S_{1,1}$, $S_{11,1}$ and $S_{12,1}$ are accessed at step 1. Instructions $S_{2,1}$, $S_{3,1}$ and $S_{20,1}$ are accessed at step 2. A solution is feasible if it does not violate any rescheduling logic rules given in Definition 2.

Fitness function: The fitness function of a chromosome is identical to the objective function of the solution. The objective is to minimize the maximum number of instructions as given in Eq. (11). All the reordering have to satisfy data dependencies and rescheduling logic rules. Otherwise the solution is checked to be infeasible. The number of access steps are fixed at the analysis step. Hence, reordering instructions to different access steps improves resource usage by reducing maximum parallelism given in Eq. (1) but average parallelism given in Eq. (4) is not altered. Thus, fitter chromosomes use lower number of ports but guarantee the same performance.

Generating new members: A new member is generated by crossover method from two parents that are chosen randomly from the population pool. The crossover happens from a randomly chosen point of the genes. Nevertheless, the crossover point should not violate any data dependencies and rescheduling logic rules. In that case, a new point is chosen and this continues until a valid crossover is achieved. We also apply single point mutation. A random gene is selected and scheduled to an available location randomly. This improves diversity among children. Nevertheless, we have observed that, the rate of mutation should be kept minimum due to data spatiality rule. Due to spatial locality, scheduling a single gene to a different access step might generate worse schedules repeatedly.

Population size: There is a trade-off in evolutionary algorithm design. Large population size provides more diversity but increases processing time whereas small population size allows to process more generations but may not generate enough diversity.

Table 2
Solution encoding of given example after recommendation step.

S1,1	S2,1	S3,1	S5,1	S6,1	S8,1	S11,1	S12,1	S15,1
1	2	2	3	3	4	1	1	4
S16,1	S17,1	S19,1	S20,1	S21,1	S24,1	S25,1	S1,2	S2,2
4	5	5	2	5	3	6	6	6

The population size is designed with two important criteria:

1. Our algorithm always operates in an evaluation window and we allow only feasible solutions in the population. Therefore data dependency and rescheduling logic rules must be applied to each solution and these operations are compute intensive
2. Let c denote the number of access steps higher than p_{ave} and the value of c is less than the evaluation window. Therefore, $\frac{c}{\alpha}$ is always less than 1

Based on given criteria, the formula for determining the population size is:

$$p(c, \alpha) = \max\{2, \ln(\alpha) * c\} \tag{12}$$

Fig. 6 shows the increase in population size with respect to α of example in Fig. 4. It is assumed that $\frac{c}{\alpha}$ is the same in all evaluation windows. In the example, we can observe from Fig. 4(b) that $\frac{c}{\alpha} = \frac{2}{6}$. Initial population is generated randomly until the population size is met.

Replacement: We admit a chromosome into the population if it is distinct and fitter than the least fit chromosome. The worst member is discarded. This improves the average fitness value of the population gradually while maintaining genetic diversity.

Termination: The algorithm terminates after $\sqrt{\alpha * c}$ successive iterations where the best objective value has not changed. From the example in Figs. 4 and 6 show the change in number of successive iterations for termination criteria with respect to α when $\frac{c}{\alpha}$ is fixed. The formula allows GA to converge to population's best result in a reasonable time.

Algorithm 2. Bypass Extraction Algorithm

Algorithm 2. Bypass Extraction Algorithm

Input: Instruction schedule produced by Maximum ILP optimization heuristic and Control Flow Graph
Output: New Instruction schedule and bypass logic block connection list

```
1 Bypass_block_list, a list that holds the connections between
  execution units, initially empty;
2 foreach Two consecutive instruction bundles in a basic block
  identified from control flow graph do
3 |  if there are load and store operations between same or
  |  another execution unit then
4 |  |  replace load and store operations with local register
  |  |  read/writes;
5 |  |  add the source and destination execution units to the
  |  |  Bypass_block_list
6 |  end
7 end
```

3.3.3. Bypass logic extraction

Bypass logic extraction algorithm replaces redundant load/store operations in basic blocks. Inside a basic block, two consecutive load/store instructions that transfer data between same or different execution units are replaced by bypass instructions and bypass blocks. Furthermore, during the execution of two consecutive VLIW instructions, data is transferred between execution units through bypass blocks instead of through memory with load/store instructions. Hence, this mechanism reduces the number of memory accesses. Bypass blocks consist of a register and a multiplexer as explained in Section 2. Bypass blocks are placed along the data paths of execution units as shown in Fig. 1, thus the latency of the bypass blocks do not take part on the critical path.

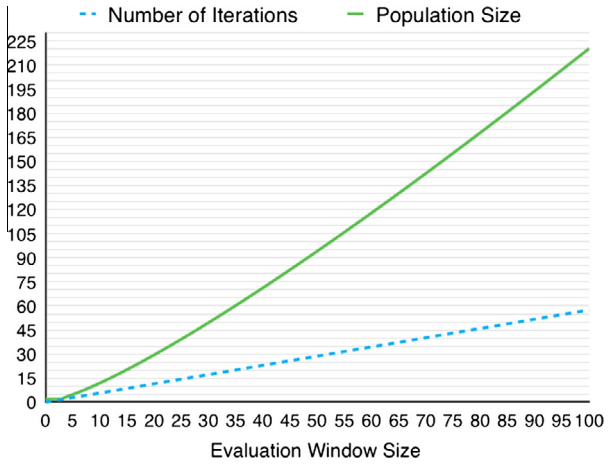

Fig. 6. The graphs show the increase in population size and the number of iterations for termination criteria for the given example in Fig. 4.

Algorithm 2 shows the bypass logic extraction algorithm. The algorithm processes the final rescheduled trace file which is produced by the maximum ILP optimization heuristic. Bypass logic extraction algorithm processes instructions in basic blocks identified by the control flow graph. In the algorithm, each basic block is traversed and consecutive instructions with load/store operations to memory are identified. Identified load/store operations are replaced with bypass instructions and execution units that execute the bypass instructions are determined for insertion of bypass blocks between these units.

4. Experimental results

In this section, we present the results of our experiments. The results are obtained by our trace based simulator [21]. Our dynamic profiler software tool uses PIN [22] binary instrumentation library in order to capture execution traces, thus all the traces consist of ×86 instructions. However, our tool is built to work with different binary instrumentation libraries. In order to show our method's versatility, we have applied our VLIW customization on FPGA and ASIC technologies, and for each technology, we use two different algorithms from different domains.

For each technology, our simulator requires the area and execution costs for each instruction. Table 3 presents the hardware costs of a single execution unit in Zynq FPGA [23] and ASIC implementation. For FPGA, we rely on our FPGA database which we hand coded in VHDL, synthesized, placed and mapped for Zynq FPGAs [3]. All units are synthesized using Xilinx Vivado 2014.4 for Kintex-7 series which is available on the Zynq FPGA and clock period is 10 ns. The dynamic power of execution unit is estimated with Vivado Power Measurement Tool. For ASIC, we have modified CACTI [24] for estimating the area of data memories in ASICs. CACTI [24] memory area estimations use following parameters, Block size 64 bytes, total size 4 GB, technology 32 nm, page size 8192 bits, burst length 8, internal prefetch width 8, input/output bus width 64, operating temperature 350 K and no selected optimizations. The area and dynamic power of non-memory operations are estimated via modifying the ARM RISC model in McPAT [25] to include the components that we use in Table 3. The RISC model is configured with 1 Ghz and uses 32 nm settings of McPAT. In order to calculate the costs of multiple execution units, the cost of a single execution unit is aggregated. The interconnection network costs are omitted. The FPGA clock period and RISC clock frequency figures are obtained from Zynq FPGA specifications [23].

In our simulator, the performance costs are measured in terms of execution cycles. However, the clock period is defined as a parameterizable input variable since hardware fabrication metrics might substantially affect the clock period of the system. Thus, our simulator can report both the execution cycle and overall execution time depending on the designer's choice. In our previous work [3], we have investigated the change of memory access speed with respect to the number of ports for our multi-port memory implementation [18] in the Zynq FPGA platform. In all our simulations, the memory instructions are assumed to take one clock cycle. The non-memory operation performance costs are taken from [26,27] for ASIC and FPGA execution units. Since, our dynamic profiler tool uses PIN binary instrumentation library [22], our traces are obtained by running the algorithms on an ×86 machine.

Memory port customization is crucial because it is an upper bound for the number of execution units. If there is not enough memory bandwidth for execution units, they become redundant for VLIW. Thus, higher number of memory ports creates bigger data path with higher number of execution units. This increases area-delay product which indicates that the VLIW implementation may get more inefficient than the RISC counterpart. On FPGAs, we have already shown that the effect of the number of read/write ports on the memory access times is not uniform [3]. We also showed that increasing number of ports degrades memory access times, and increasing the number of write ports degrades the memory performance more than increasing the number of read ports [3]. This is due to the fact that, in order to realize multiple write ports, extra memory banks are required. The data in the banks are connected with multiplexers that increase the delay as the number of banks increases, thus lowering the access speeds.

We have applied our VLIW customization method on two different algorithms. These are string matching and BLAS algorithms [28]. Our dynamic profiling tool explained in Section 3.1 has generated the traces. String matching has been studied extensively and a significant amount of algorithms has already been proposed [29]. Benchmark algorithms are shown in Table 4, which also shows the main search method of each string matching algorithm. Although, there are different methods, all of them are highly memory intensive [29]. Input set of string matching algorithms contains one centimorgan DNA base pairs which is approximately one million characters of text. In this text string, we search for a

Table 3
Resource usage and power consumption estimations of the execution unit in Zynq FPGA and ASIC.

Unit	FPGA resources					ASIC resources	
	LUT	FF	Mux	DSP48e	Power (W)	Area (mm^2)	Power (W)
Floating point unit	554	720	110	–	1.7	1.2	0.848
Arithmetic logic unit	80	350	–	1	0.971	0.25	0.04
Load/store unit	8	32	32	–	0.580	0.11	0.0024
Branch unit	34	32	32	–	0.476	0.1	0.0022
Division	1250	3200	1100	–	8.5	0.32	0.08

Table 4
String matching algorithms and abbreviations.

Abbreviation	Algorithm	Type
FSBNDMQ	Forward Simplified Backward Nondeterministic DAWG Matching with q-grams	Bit-parallel
BMH-SBNDM	Backward Nondeterministic DAWG Matching with Horspool Shift	Bit-parallel
KBNDM	Factorized Backward Nondeterministic DAWG Matching	Bit-parallel
FAOSO	Fast Average Optimal Shift Or	Bit-parallel
SEBOM	Simplified Extended Backward Oracle Matching	Automata
FBOM	Forward Backward Oracle Matching	Automata
SFBOM	Simplified Forward Backward Oracle Matching	Automata
TVSBS	TVSBS: A Fast Exact Pattern Matching Algorithm for Biological Sequences	Comparison
FJS	Franek Jennings Smyth String Matching	Comparison
GRASPM	Genomic Rapid Algorithm for String Pattern Matching	Comparison

pattern with a length of ten thousand characters and each character is one byte, hence a pattern is 10 kB long. The text alphabet size is four.

BLAS [30] benchmarks have been one of de facto benchmarks from high performance computing to embedded systems. BLAS benchmarks include numerical, linear, scalar, vector, vector–vector, matrix–vector and matrix–matrix operations. Today, different application domains necessitate execution of complicated linear algebra programs which make use of a few different low level operations. Hence, every improvement in those low level operations make significant impact on the overall application performance. BLAS algorithms are more compute-intensive than string matching operations. Therefore, wider parallelism will require more complicated ALU units which are suitable for experimenting on ASIC technologies. BLAS input data has the following properties. We set the size of input matrices to 100 in all matrix–matrix operations, 200×100 to matrix–vector operations. Rest of the arrays are modified to 1000 elements and benchmarks with scalars are modified to iterate 1000 times. Input data is assumed to fit in the memory.

In our simulator, we have assumed that loading data from external memory to FPGA memory is handled by a separate buffering mechanism, which controls a multi-paged memory architecture. When one page is consumed, data handling mechanism switches to the other page which is already loaded with data. Thus, memory references take constant amount of cycles. Xilinx Zynq family has up to 3020 kB of memory [23], therefore our paged memory architecture can load multiple pages which will hide the latency of loading patterns from BRAMs. The amount of buffer pages can be allocated according to the requirements of the input data and available memory aside from the processor implementation.

We have evaluated the performance and efficiency of our tool in two sets of experiments. In the first set, we compared our method with a recent graph-based force-directed parallelism estimation method of Jordans et al. (JPE) [16]. This evaluation is explained in Section 4.1. In the second set, we compare the performance of our customized VLIW processor model with a RISC processor model. RISC processors are widely used in embedded systems and they are freely available on FPGA systems. Altera Nios-II [31] and Xilinx Microblaze [32] are two of the most important soft RISC processors. Both of them can be customized in order to have different sizes of caches, arithmetic units, etc. In maximum customizations, they are connected to the true dual ports of block rams (BRAMs) inside the FPGAs. Thus, our customized VLIW is compared with a RISC processor model which is the de facto baseline architecture. This is explained in Sections 4.2 and 4.3. We do not generate any code for the customized VLIW, therefore comparing our reference VLIW model with a VLIW processor is out of scope of this paper. Our VLIW customization can be used as the starting point for customizable VLIW processors.

4.1. A comparative study of VLIW customization

We have extracted the longest basic block traces in string matching and BLAS algorithms because the JPE algorithm only works with basic blocks. Since in [16], authors have explained that the selection of scheduling algorithm is independent from parallelism estimation method, we have used our rescheduling logic rules given in Section 3.1 so as to make an objective comparison between estimation methods. If we had used another scheduling algorithm, we would not be able to understand whether the improvement is due to the estimation method or the scheduling method. We apply JPE and our algorithm to selected applications. Then, we extract the required number of read/write ports and execution units calculated by both algorithms.

In Tables 5 and 6, we compare the best memory configurations, hardware usage and power consumption of execution units when both deliver the same performance in the selected benchmarks that are shown in Figs. 7 and 9. Thus, in order to provide same execution performance, JPE requires more components in the execution units and higher number of memory ports. Table 5 also presents the number of memory ports and the number of nodes in the basic blocks (BB) for clarity. Higher number of ports increases both the FPGA/ASIC memory area and power consumption. Selected test cases from string matching and BLAS show that our method can suggest better or the same configurations than JPE in all test cases. In most of the benchmarks, we have observed that when our method decreases the number of ports compared JPE, these reductions of memory ports correspond to LD/ST or BR units in the execution units.

Table 7 compares the solution finding times of JPE algorithm and ours. We have observed that our algorithm can converge to a solution faster than JPE, because our GA method can explore and rule out worse schedules with the help of mutation operator. However, for graphs with very large number of basic block nodes, our algorithm spends more time due to increased population size and termination criteria of the GA.

4.2. Results of VLIW customization of string matching algorithms with FPGA technology parameters

All our multi-port and true dual port (TDP) memory configurations have 32 bits word size and depth is 4096. With the given memory configuration and development platform, 2R 1W true dual port memory consumes 16 BRAMs and 1 LUT on Xilinx FPGAs. Table 8 presents the TDP RISC processor model components that we have used in the simulator. RISC model has been simulated with the perfect cache behavior. Our VLIW has performed much better with cache misses enabled in our simulator. Therefore, we have opted to present the worst case improvements of our VLIW versus all RISC core configurations.

We present the experimental results of string matching algorithms in Figs. 7 and 8. Our custom VLIW model runs 3× faster on average than the RISC model. Given our dataset, one of bit-parallel string matching algorithms, FSBNDM has one of the best performance among all the algorithms. Bit-parallel string matching algorithms have exploited the multi-port very efficiently and executed much faster than comparison and automata based algorithms. This is because the dominant operators in bit-parallel algorithms are shift-and operations. Moreover, their

Table 5
Resource and power consumption of memories that are recommended by Jordans et al. [16] and our method.

Benchmark	BB nodes	Jordans et al.						Ours					
		Mem.	FPGA			ASIC		Mem.	FPGA			ASIC	
		R, W	BRAM	LUT	Power (W)	Area (mm^2)	Power (W)	R, W	BRAM	LUT	Power (W)	Area (mm^2)	Power (W)
FSNBDM	28	4R, 1W	31	1	0.243	901	0.26	3R, 1W	24	1	0.220	621	0.13
SEBOM	31	3R, 1W	64	57	0.220	621	0.13	3R, 1W	64	57	0.220	621	0.13
FJS	63	4R, 2W	64	57	0.297	939	0.28	3R, 2W	48	49	0.275	801	0.21
FAOSO	113	3R, 1W	24	1	0.220	525	0.13	3R, 1W	24	1	0.220	621	0.13
TVSBS	124	4R, 2W	64	57	0.297	939	0.28	3R, 2W	48	49	0.275	801	0.21
BMH-SBNDM	36	5R, 2W	80	64	0.3200	1155	0.38	4R, 1W	31	1	0.243	901	0.26
CAXPY	48	3R, 3W	72	86	0.305	1003	0.26	3R, 2W	48	49	0.275	801	0.21
CCOPY	46	6R, 4W	192	174	0.401	2303	0.39	4R, 4W	127	98	0.356	1544	0.37
CSSCAL	16	3R, 2W	48	49	0.275	801	0.21	2R, 2W	31	23	0.215	572	0.18
DGEMM	60	4R, 1W	31	1	0.243	901	0.26	3R, 1W	24	1	0.220	621	0.13
DGER	46	3R, 1W	24	1	0.220	525	0.13	3R, 1W	24	1	0.220	621	0.13
SCASUM	19	8R, 2W	128	162	0.407	2633	0.30	8R, 1W	64	1	0.369	2279	0.23
CSROT	51	4R, 4W	127	98	0.357	1544	0.37	3R, 3W	72	86	0.305	1003	0.37
DTRSM	54	6R, 1W	56	1	0.289	1518	0.20	5R, 1W	48	1	0.266	1191	0.18

Table 6
Resource consumption of execution units and Jordans et al.'s (JPE) redundant hardware compared to ours.

	Jordans et al. [16]	Ours	JPE's redundant HW
FSNBDM	3 * ALU, 2 * BR, 3 * LD/ST	3 * ALU, 2 * BR, 2 * LD/ST	1 * LD/ST
SEBOM	3 * ALU, 2 * BR, 2 * LD/ST	3 * ALU, 2 * BR, 2 * LD/ST	–
FJS	3 * ALU, 2 * BR, 4 * LD/ST	3 * ALU, 2 * BR, 3 * LD/ST	1 * LD/ST
FAOSO	3 * ALU, 2 * BR, 2 * LD/ST	3 * ALU, 2 * BR, 2 * LD/ST	–
TVSBS	3 * ALU, 2 * BR, 4 * LD/ST	3 * ALU, 2 * BR, 3 * LD/ST	1 * LD/ST
BMH-SBNDM	4 * ALU, 3 * BR, 3 * LD/ST	3 * ALU, 2 * BR, 3 * LD/ST	1 * ALU, 1 * BR
CAXPY	3 * FP, 3 * ALU, 3 * BR, 3 * LD/ST	3 * FP, 3 * ALU, 1 * BR, 2 * LD/ST	1 * BR
CCOPY	2 * ALU, 6 * BR, 6 * LD/ST	2 * ALU, 4 * BR, 4 * LD/ST	1 * BR, 1 * LD/ST
CSSCAL	3 * FP, 3 * ALU, 2 * BR, 3 * LD/ST	3 * FP, 3 * ALU, 2 * BR, 2 * LD/ST	1 * LD/ST
DGEMM	3 * FP, 3 * ALU, 2 * BR, 3 * LD/ST	3 * FP, 3 * ALU, 2 * BR, 2 * LD/ST	1 * BR
DGER	3 * FP, 3 * ALU, 2 * BR, 2 * LD/ST	3 * FP, 3 * ALU, 2 * BR, 2 * LD/ST	–
SCASUM	4 * FP, 4 * ALU, 2 * BR, 6 * LD/ST	3 * FP, 3 * ALU, 2 * BR, 6 * LD/ST	1 * FP, 1 * ALU
CSROT	4 * FP, 4 * ALU, 2 * BR, 4 * LD/ST	3 * FP, 3 * ALU, 2 * BR, 3 * LD/ST	1 * FP, 1 * ALU, 1 * LD/ST
DTRSM	2 * FP, 2 * ALU, 2 * BR, 4 * LD/ST	2 * FP, 2 * ALU, 2 * BR, 3 * LD/ST	1 * LD/ST

implementations allow full utilization of memory bandwidth in most of the algorithms' running time. In contrast, automata algorithms heavily use branch logic and comparison algorithms use multiple cycle operators such as multiplication and division. These two reasons prevent automata and comparison based algorithms from achieving high speed ups.

Fig. 8 also presents the area-delay product of customized VLIW multi-port configurations which are normalized to RISC with True Dual Port memory (RISC TDP). Results show that majority of multi-port configurations are more efficient than RISC TDP implementations. However, algorithms with multiple write ports can get inefficient due to the excessive usage of BRAM and LUT. Table 5 presents the consumption of BRAM and LUT blocks of VLIW data memory. Number of BRAMS and LUTs increase with increasing read and write ports. Fig. 7 also shows the reduction of memory operations after bypass extraction algorithm is applied. The algorithm has managed to reduce load/store operations between 5% and 10%.

Fig. 8 shows the VLIW memory configurations. We have observed that read/write port numbers and average parallelism of an algorithm do not necessarily provide the performance hint. For example, FSBNDM algorithm has the least average parallelism among all algorithms. TVSBS has one of the largest number of read/write ports and highest average parallelism. However, it is much slower than FSBNDM. This is due to the fact that FSBNDM implementation is more efficient in TVSBS algorithm. The main reason is TVSBS employs integer division operation and this slows down its performance.

When we compare the average power consumptions of RISC and VLIW, Fig. 8 shows that VLIW is more power hungry. The amount of work performed by VLIW is more than the RISC counterpart. To reduce the power consumption, one can reduce the clock frequency of the VLIW system implementation which will reduce the power consumption.

4.3. Results of VLIW customization of BLAS algorithms with ASIC technology parameters

BLAS results are shown in Figs. 9 and 10. Table 9 presents the TDP RISC processor model components that we have used in the simulator. Experiments have shown that benchmarks that are inherently parallel are parallelized eminently. We have observed that significant performance gains are observed when either given algorithm's average parallelism is high or dominant operation of the algorithm is a costly operation such as division operation.

Exploiting available parallelism with available memory ports provides performance advantage over RISC. Another advantage occurs when dominant operation is a costly operations because multiple Execution Units of the VLIW can instantiate this operation.

Fig. 9 also presents the area-delay product of the multi-port configuration which are normalized to dual port configurations. All of the benchmarks except two are more efficient than dual port configuration. Unlike string matching algorithms on FPGA, algorithms with multiple write ports on ASIC are more efficient. This

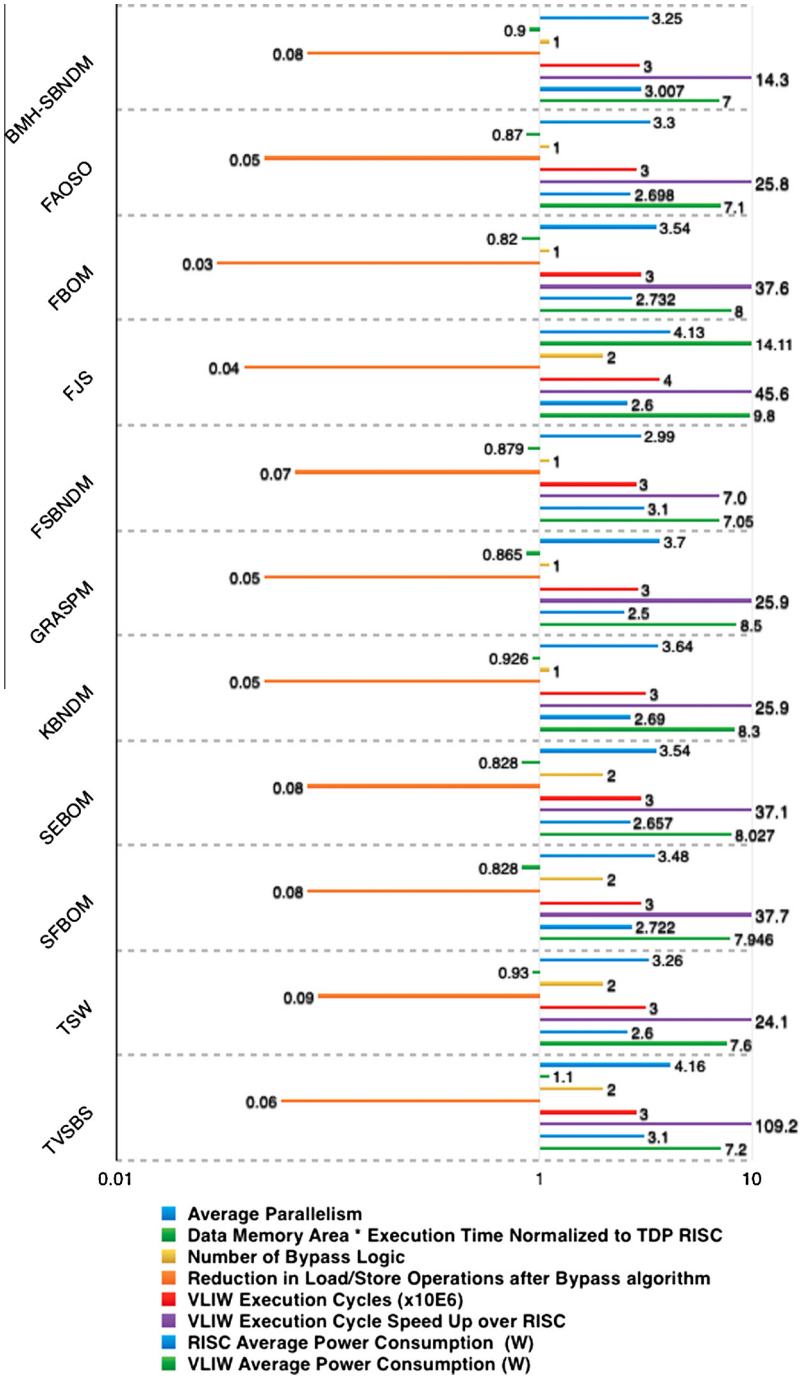

Fig. 7. Custom VLIW compared to RISC core with FPGA cost parameters for string matching algorithms.

is because additional logic introduced by multiple write ports are handled more effectively and inherent speed of ASICs are more than FPGAs. In the area-delay product measurements, we have not included the areas of execution units because it has prevented to observe the change in performance when memory area differs. Similarly, we have not included the instruction memory in this work because instruction memory can also depend mostly the instruction selection during the compile time.

The number of ports hints the success of our method in extracting performance compared to RISC. From Fig. 10, ICAMAX and DGEMM can be identified as two of the least port consuming algorithms. Nevertheless, they improve the execution time 4× on average compared to RISC model.

Fig. 10 shows that DGEMV, DTRMV and SCASUM benchmarks have generated much higher number of port sizes than the average parallelism value. This could be overcome by increasing the evaluation windows size, α. Yet, larger evaluation window increases the design space and solution finding time drastically.

It is also shown in Fig. 10 that CSWAP, ICAMAX and CCOPY have memory configurations which are less than the average parallelism. In these benchmarks, GA has found suitable schedules where Signature Removal Rule explained in Definition 3 is applied. Therefore, unnecessary memory read instructions are deleted, hence better memory configurations are produced.

Bypass logic has managed to reduce load/store operations up to 11% among BLAS algorithms. For algorithms which have higher

Table 7
Comparison of runtime of ours and Jordans et al. [16].

Benchmark	Jordans et al. [16] Cycles	Ours Cycles	Our imp.
FSNBDM	27,520	4320	6.3×
SEBOM	21,780	4780	4.5×
FJS	40,810	103,000	−2.5×
FAOSO	128,150	1,107,020	−8.6×
TVSBS	260,360	1,317,250	−5×
BMH-SBNDM	80,640	20,130	4×
CAXPY	100,260	64,400	1.5×
CCOPY	45,960	31,040	1.5×
CSSCAL	16,820	1330	12×
DGEMM	133,560	98,090	1.3×
DGER	130,390	31,040	4.2×
SCASUM	13,360	1580	8.4×
CSROT	205,920	68,430	3×
DTRSM	76,280	72,840	1.05×

parallelism in memory instructions get significant advantages, because multiple bypass logic blocks could be placed, when multiple data forwarding is detected. This has allowed to create an extra slot for scheduling multiple operations.

Average power consumption drastically increases when VLIW exploits the parallelism in the algorithms as shown in Fig. 9. Algorithms which are inherently parallel like CCOPY also consume a lot of power due to increasing capacitance of larger memory and execution units. Reducing the number of memory ports and execution units can decrease the power consumption.

As an example, one of the customized VLIW reference model is shown in Fig. 11. It is the output of FSBNDM string matching algorithm. It has three read ports and one write port. There are three execution units. Functional Units 1 and 2 are exactly the same. Each FU consists of an ALU, Branch Logic and Load/Store Unit. Functional Unit 3 has only one ALU. Bypass logic connects the output of Functional Unit 1 to the input of Functional Unit 2.

5. Related work

For the last three decades, several studies have been performed on VLIW customization. Exploring available ILP from a given program has been crucial for application specific VLIW processors in order to reduce compiler effort and prevent redundant hardware. State of the art ILP extraction algorithms are based on either instruction traces [33,6,4,5,34–37] or dependency graphs [38,39,15,40–42].

The most important distinctions between compile time methods and our method are twofold. First of all at the compile time, the compiler needs the details of the memory and execution units such as the number of memory ports and the composition of execution units in order to apply scheduling heuristics such as list scheduling. Hence, this is the reason why the state-of-the-art study from Jordans et al. [16] uses distribution graphs to have a priority resource constraint for their list scheduling algorithm. Secondly, at the compile time data dependencies are not extracted fully. For this reason, many compute intensive data dependency analysis methods were suggested.

In more detail, in order to apply dependency checking and structural hazards at the compile time, compilers apply several optimizations. All the hazards must be figured out at the compile time. Software pipelining, trace scheduling, predicated execution and speculative execution have been major compiler optimization for improving ILP. Trace scheduling [43] required additional code when operations are reordered. Speculative execution [44] helps

Table 8
The area and power consumption of TDP RISC model used in our FPGA simulations.

	Execution unit		Data memory	
	Area (mm^2) 1 ∗ FP, 1 ∗ ALU, 1 ∗ BR, 1 ∗ LD/ST	Power (W)	Area (mm^2) 2R, 1W	Power (W)
Without division	676 ∗ LUT, 1810 ∗ FF, 174 ∗ Mux, 1 ∗ DSP48e	3.7	16 ∗ BRAM, 1 ∗ LUT	0.198
With division	1926 ∗ LUT, 5010 ∗ FF, 1274 ∗ Mux, 1 ∗ DSP48e	12.2	16 ∗ BRAM, 1 ∗ LUT	0.198

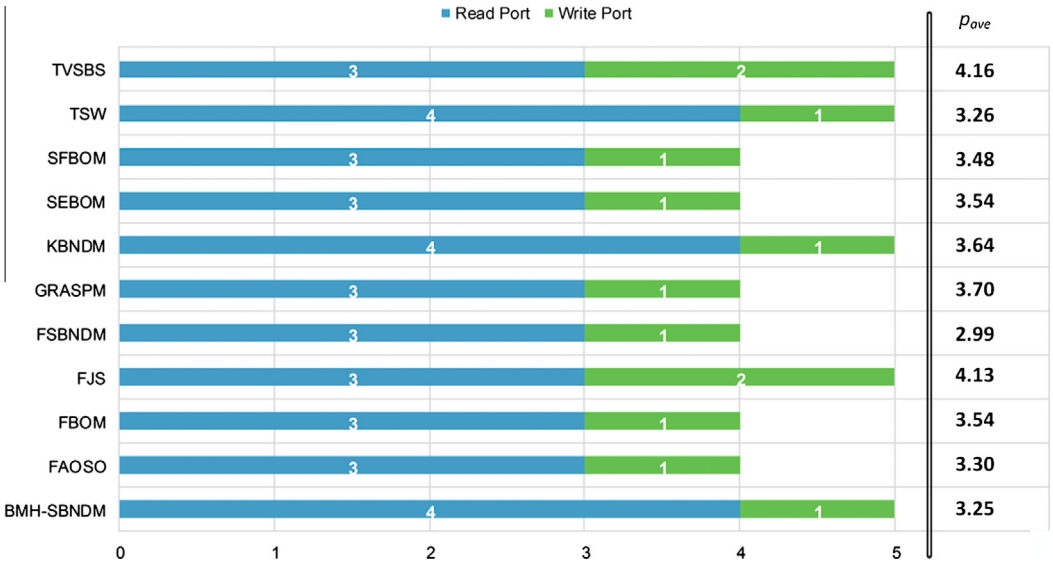

Fig. 8. Number of ports of the multi port memory for string matching algorithms.

DOI : 10.1016/j.micpro.2015.09.005

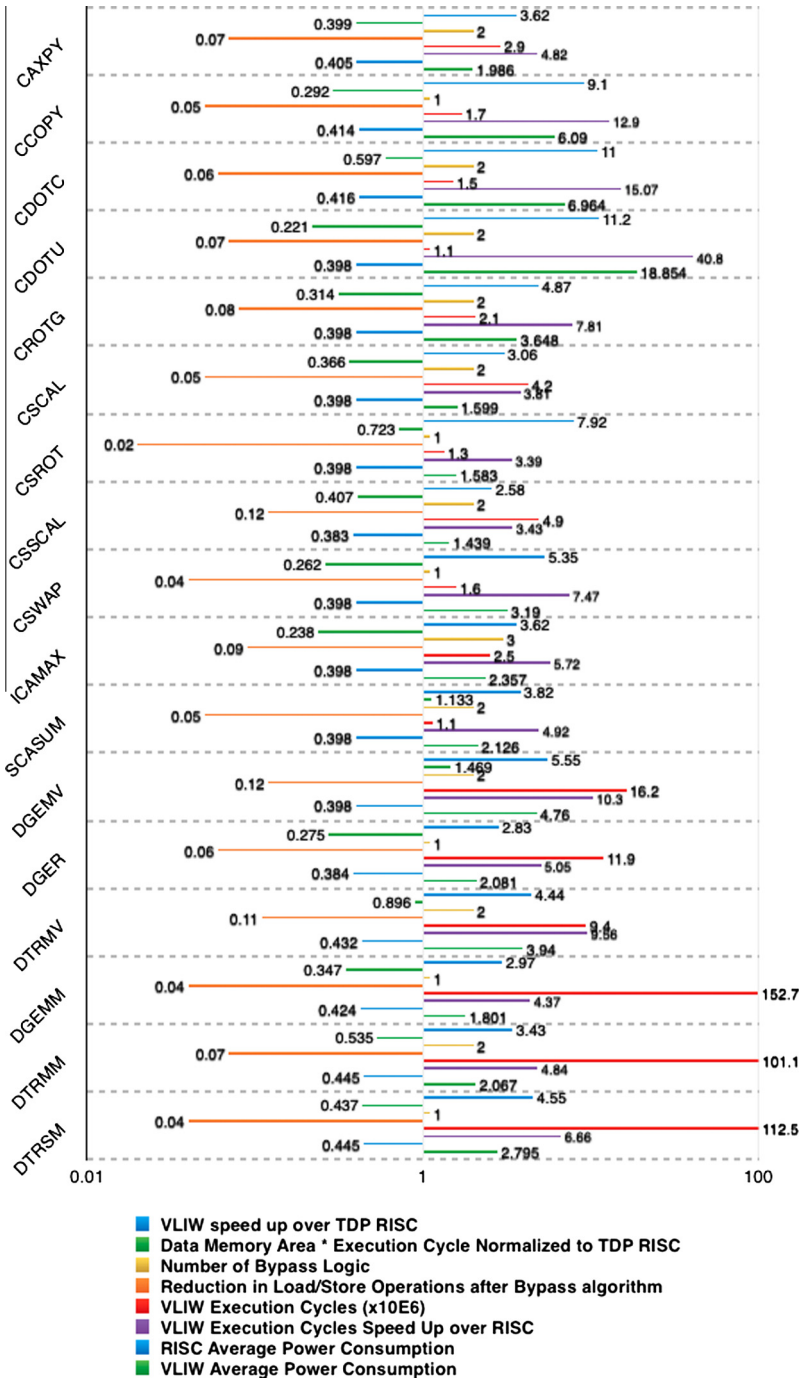

Fig. 9. Custom VLIW compared to RISC core with ASIC cost parameters for BLAS algorithms.

the reduction of compensation code and moves instructions over the branches. The speculatively executed code should not produce any stalls to the processor pipeline when it is not needed. Speculative execution can be implemented as hardware or software. Software pipelining [45] aims at compacting loop kernels by minimizing initiation intervals. Hierarchical reduction [45] is the method to simplify scheduling process by compacting and representing scheduled program components as a single component. These scheduled components preserve and expose scheduling constraints to the compiler and continues until all components are reduced to a single program node. Approaches that extract ILP parallelism that do not work with execution traces require memory data dependency analysis methods such as Omega test [46], GCD test [47] and points-to-analysis [48] which are computationally expensive.

In contrary to previous compile time methods, our Genetic Algorithm (GA) that is explained in Section 3.3.2 analyzes execution traces where all the address and register dependencies are followed from real memory addresses and registers. Therefore, all of the data dependency and structural hazards can be solved. Moreover, by the time our algorithms execute, the information such as the number of memory ports and execution units are unknown. Nevertheless, our method is designed to generate aforementioned information.

Previous works on VLIW customization have discussed that due to large design exploration space, parallelism information should

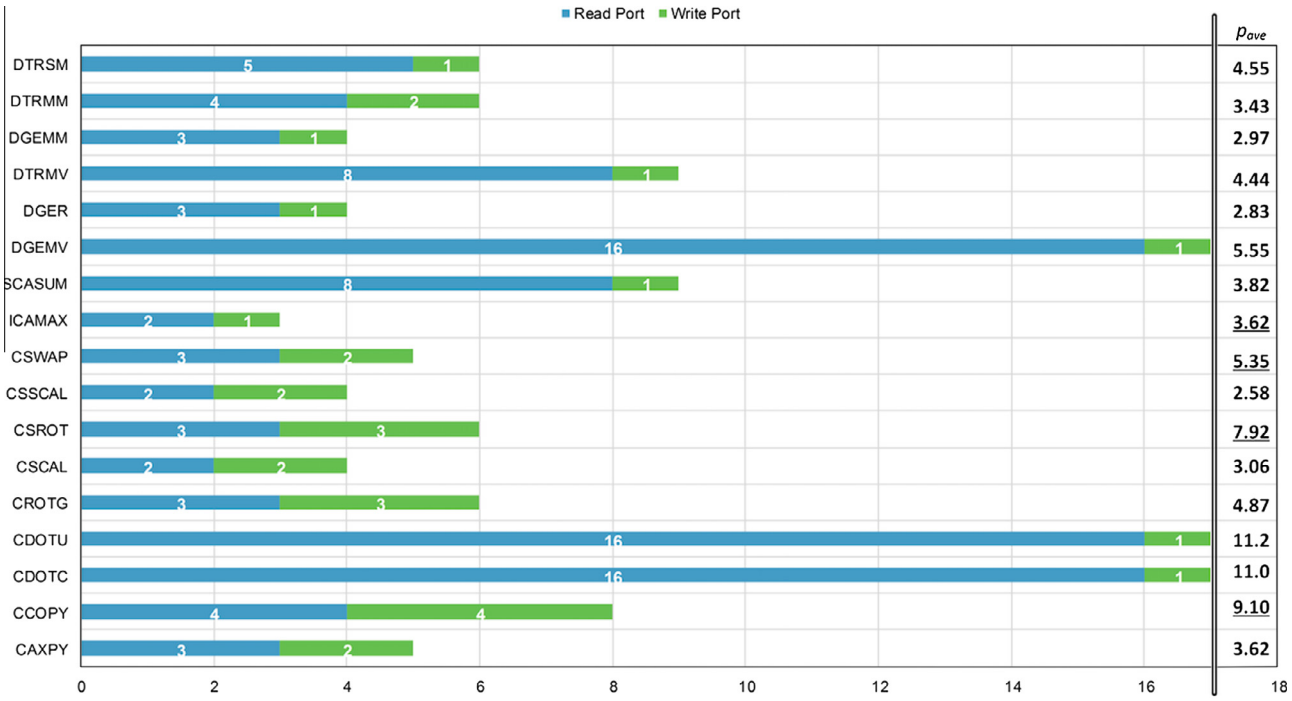

Fig. 10. Number of ports of the multi port memory for BLAS algorithms.

Table 9
The area and power consumption of TDP RISC model used in our ASIC simulations.

	Execution unit		Data memory	
	Area (mm^2) $1 * FP, 1 * ALU, 1 * BR, 1 * LD/ST$	Power (W)	Area (mm^2) 2R, 1 W	Power (W)
Without division	1.66	0.9	493	0.12
With division	1.98	0.98	493	0.12

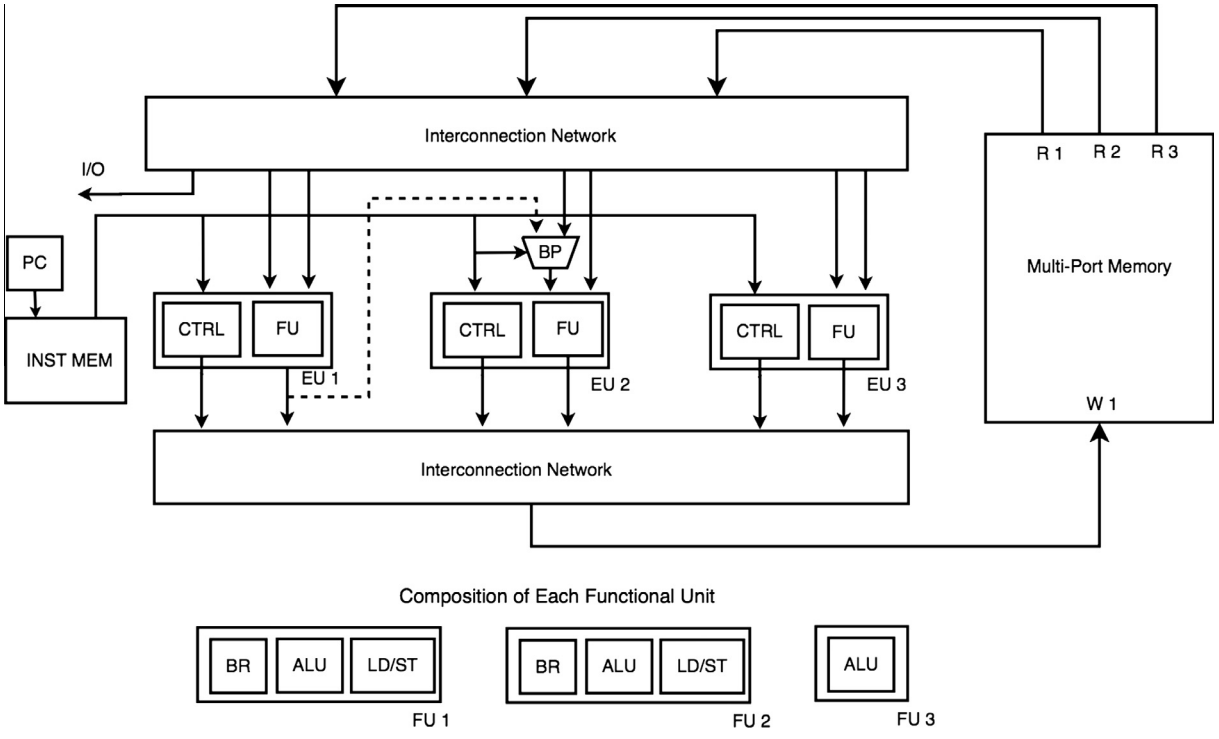

Fig. 11. Customized VLIW architecture with three execution units and composition of each functional unit.

DOI : 10.1016/j.micpro.2015.09.005

only be used on the most relevant parameters of VLIWs. Therefore, traditionally, data path customization of VLIWs has been studied extensively [42]. A meager number of studies has also investigated the cache geometries which is essentially the generic cache organization problem that is coinciding with general purpose superscalar processor cache design [49,50,40]. Instead, we tackle both tackle VLIW data path by deciding the number and composition of execution units and the number of memory ports which provide sufficient memory bandwidth to execution units.

Recent work [51] considers finding the efficient number of VLIW execution clusters while keeping memory and register file topology unchanged. The memory bandwidth which is consumed by selected execution units are assumed to be supported by available memory topology, thus the number of memory ports are fixed. In contrast, our method finds the required memory number of memory ports which guarantees the memory bandwidth and the combination of functional units. Therefore, the data path, namely the number and composition of execution units and their compositions guarantee not to exceed the available memory bandwidth.

Trace based design space exploration in [52] extracts and schedules only non-memory instructions from a given architecture. Thus, the scheduler must be provided with the number of memory ports and number of functional units as templates. However, our method do not need templates. It can generate any combination of execution units and number of memory ports from the application. Our default exploration method is bounded ILP extraction. However, an upper bound on number of execution units and memory ports can be given.

Moon and Ebcioğlu [36] have performed an empirical study characterizing suitable memory ports in VLIW processors. They have identified several VLIW templates of ALU, memory configurations. They measure ILP and execution speed up. Nevertheless, selected templates are not tailored for a given application and the composition of execution units is not selected. Execution units are identical ALU units. In addition, memory ports are selected based on maximum parallelism extracted from the given trace. However, coarse grain execution unit selection and maximum parallelism based VLIW designs result in poor memory utilization when the number of memory ports increase [53]. Our method uses average parallelism and maximum parallelism, tailors the composition of execution units and the number of memory ports for a given application. Thus, our approach yield better memory utilization.

A recent study has presented a modified list scheduling to extract parallelism for sharing VLIW register ports [54]. Authors have focused on adding more constraints to list scheduling algorithm for preventing resource contentions between different execution units while keeping performance degradation at low levels. However, unlike our work memory operations and non-memory operations haven't been treated separately which can relax constraints significantly.

Jordans et al. [16] have studied different parallelism estimation methods. When applied on a modified version of list scheduling algorithm, force based parallelism have provided more accurate estimation of ILP than maximum and average parallelism when they are coupled with binary search based estimation strategies. Our method extracts parallelism information from dynamically profiled execution traces as opposed to static profiling, therefore it does not require further estimation strategies because all the control flow is extracted from the trace. Similarly, when working with traces, calculating distribution graph based parallelism estimation methods add extra computation cost. Therefore, average parallelism estimation methods tend to be more suitable.

Nicolau and Fisher have showed the capabilities of VLIW hardware by measuring available (ILP) and applied trace scheduling [4,5]. They measured execution traces and show speedups of 1000 for an ideal machine which has infinite hardware resources, perfect branch prediction, perfect address disambiguation. Similarly, Liao and Wolfe [6] has studied how VLIWs are suitable in video applications domain. They have used trace driven simulation to evaluate video applications. Traces are scheduled and average parallelism is measured under ideal machine conditions and all operations are single cycle operations. On the other hand, our method does not require that "ideal machine" to perform. Our ILP extraction method works with instruction traces which are profiled from any given machine architecture.

There have been studies to extract ILP from a program for a given VLIW architecture. The authors in [35] explored ILP for an 8-way VLIW with different execution unit templates. Instruction traces are scheduled in order to extract basic block and branch statistics, data sizes, working set sizes, and average parallelism. Authors have found that block level scheduling has not provided enough parallelism for 8-issue parallelism. Similarly, the results in [34] present that scheduling beyond basic blocks can support performance which is more than two instructions per cycle on average. Nevertheless, this performance is only possible if necessary memory bandwidth is provided. Authors have used average parallelism for their parallelism metrics. On the contrary, we apply global instruction scheduling. Our method extracts and schedules memory and non-memory operations separately. In the beginning, memory instructions are scheduled without any resource constraints. In this way, maximal memory bandwidth is extracted and the number of memory ports are chosen. Then, the number of memory ports is used as a constraint and non-memory operations are scheduled. Hence, the overall schedule of non-memory operations provide the necessary functional units.

Smoothability metric presented by Theobald et al. presents a metric for how evenly the parallel portions of applications are distributed [33]. Their work has shown that smoothability can be achieved by either the given application's parallelism is evenly distributed or the underlying architecture and the scheduler can provide enough parallelism to increase performance by scheduling parallel instructions. However, they do not consider how memory operations and non-memory operations are distributed over applications. Detection for smoothability requires multiple runs, because it is architecture specific. Different architectures or data inputs yield different results. Our method utilizes maximum parallelism, average parallelism and memory utilization metrics in order to characterize the given application.

Lam and Wilson's study of instruction traces which consist of many branches has showed that parallelism could be seriously limited by memory address ambiguity and control dependency during compile time [37]. Similarly Fisher et al. have showed that ILP extraction with global scheduling can provide significant performance improvements over superscalar RISC [43,45]. Hence, software pipelining and trace scheduling have become the major compiler optimizations for VLIWs. Software pipelining [45,55] aims at compacting loop kernels by minimizing initiation intervals. Trace scheduling [43] extracted and scheduled highly probable traces beyond basic blocks. As opposed to ILP extraction for code generation methods, our method works provides a fast way to obtain a single design point for VLIW customization. Thus, it can easily be coupled with existing design space exploration frameworks for finding the pareto-design curve of underlying architecture [40,41].

6. Conclusion

In this paper, a method for VLIW customization was presented. The success of a VLIW customization method is dependent on its capability of extracting the existing ILP from a given algorithm. Hence, while designing our VLIW customization method, we have made three design decisions for maximizing ILP extraction. First of

all, we have chosen to work with execution traces that have allowed us to capture the exact data flow and control flow of the given algorithm. Second, we have differentiated memory instructions and non-memory instructions because customizing VLIW multi-port memory from memory operations have provided better memory utilization. Moreover, processing non-memory operations for data path exploration has allowed us to increase performance and memory efficiency. Lastly, we have designed a genetic algorithm that processes execution traces in evaluation windows to cope with large design exploration spaces. Our method based on the aforementioned design decisions has similar or more compact customized VLIW processor configurations than the state-of-the-art method on selected benchmarks. In addition, performance, power consumption and area-delay product metrics have showed that our customized VLIW models are faster and more efficient than RISC processor models.

Acknowledgement

This work is supported by the Turkish Ministry of Development under the TAM Project, Number 2007K120610.

References

[1] J.A. Fisher, P. Faraboschi, C. Young, Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools, Elsevier, 2005.
[2] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, F. Homewood, Lx: A Technology Platform for Customizable VLIW Embedded Processing, vol. 28, ACM, 2000.
[3] G.A. Malazgirt, H.E. Yantir, A. Yurdakul, S. Niar, Application specific multi-port memory customization in FPGAs, in: 24th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2014, pp. 1–4.
[4] A. Nicolau, J.A. Fisher, Measuring the parallelism available for very long instruction word architectures, IEEE Trans. Comput. 33 (11) (1984) 968–976.
[5] A. Nicolau, J.A. Fisher, Using an oracle to measure potential parallelism in single instruction stream programs, in: Proceedings of the 14th Annual Workshop on Microprogramming, MICRO 14, 1981, pp. 171–182.
[6] H. Liao, A. Wolfe, Available parallelism in video applications, in: Proceedings. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, IEEE, 1997, pp. 321–329.
[7] A.D. Samples, Profile-driven Compilation, Tech. Rep., Berkeley, CA, USA, 1991.
[8] M. Gort, J. Anderson, Range and bitmask analysis for hardware optimization in high-level synthesis, in: 18th Asia and South Pacific Design Automation Conference (ASP-DAC), 2013, pp. 773–779.
[9] Z. Yuan, Y. Ma, J. Bian, K. Zhao, Automatic enhanced CDFG generation based on runtime instrumentation, in: IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD), IEEE, 2013, pp. 92–97.
[10] S. Wong, T. Van As, G. Brown, ρ-VEX: a reconfigurable and extensible softcore VLIW processor, in: ICECE Technology, 2008, FPT 2008. International Conference on, IEEE, 2008, pp. 369–372.
[11] M. Purnaprajna, P. Ienne, Making wide-issue VLIW processors viable on FPGAs, ACM Trans. Archit. Code Optim. 8 (4) (2012) 33:1–33:16.
[12] W.-T. Shiue, C. Chakrabarti, Multi-module multi-port memory design for low power embedded systems, Des. Automat. Embed. Syst. 9 (4) (2004) 235–261.
[13] V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D.C. Cronquist, M. Sivaraman, PICO: automatically designing custom computers, Computer 35 (9) (2002) 39–47.
[14] P. Salmela, R. Makinen, P. Jaaskelainen, J. Takala, Loop scheduling for transport triggered architecture processors, in: International Symposium on System-on-Chip, 2006, pp. 1–4.
[15] B.R. Rau, V. Kathail, S. Aditya, Machine-description Driven Compilers for EPIC Processors, Hewlett Packard Laboratories, 1998.
[16] R. Jordans, R. Corvino, L. Józwiak, H. Corporaal, Exploring processor parallelism: estimation methods and optimization strategies, in: IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013, pp. 18–23.
[17] S.-M. Moon, S.D. Carson, Generalized multiway branch unit for VLIW microprocessors, IEEE Trans. Parallel Distrib. Syst. 6 (8) (1995) 850–862.
[18] H.E. Yantir, A. Yurdakul, An efficient heterogeneous register file implementation for FPGAs, in: IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2014, pp. 293–298.
[19] A.V. Aho, Compilers: Principles, Techniques and Tools, 2/e, Pearson, 2003.
[20] O. Alp, E. Erkut, Z. Drezner, An efficient genetic algorithm for the p-median problem, Ann. Oper. Res. 122 (1–4) (2003).
[21] G.A. Malazgirt, A. Yurdakul, S. Niar, Mipt: rapid exploration and evaluation for migrating sequential algorithms to multiprocessing systems with multi-port memories, in: International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2014, pp. 776–783.
[22] C.-K. Luk, R. Cohn, Pin: building customized program analysis tools with dynamic instrumentation, ACM SIGPLAN Notices, vol. 40, ACM, 2005.
[23] Xilinx, Zynq-7000 All Programmable SoC Technical Reference Manual.
[24] N. Muralimanohar, R. Balasubramonian, N.P. Jouppi, Cacti 6.0: A Tool to Model Large Caches, HP Laboratories.
[25] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, N.P. Jouppi, The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area, and timing, ACM (TACO) 10 (1) (2013).
[26] A. Fog, Instruction Tables. <http://www.agner.org/optimize/instructiontables.pdf>.
[27] Intel Architectures Software Manual. <http://goo.gl/5yvvrt> (accessed 30.05.15).
[28] Basic Linear Algebra Subprograms. <http://www.netlib.org/blas> (accessed 30.05.15).
[29] S. Faro, T. Lecroq, The exact online string matching problem: a review of the most recent results, ACM Comput. Surv. 45 (2) (2013) 13:1–13:42.
[30] C.L. Lawson, R.J. Hanson, D.R. Kincaid, F.T. Krogh, Basic linear algebra subprograms for Fortran usage, ACM Trans. Math. Softw. (TOMS) 5 (3) (1979) 308–323.
[31] I. NIOS, Processor Reference Handbook, Altera Corporation, 2008.
[32] I. Xilinx, Microblaze Processor Reference Guide, Reference Manual, 2006, pp. 23.
[33] K.B. Theobald, G.R. Gao, L.J. Hendren, On the limits of program parallelism and its smoothability, SIGMICRO Newsl. 23 (1–2) (1992) 10–19.
[34] M.D. Smith, M. Johnson, M.A. Horowitz, Limits on multiple instruction issue, in: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS III, 1989, pp. 290–302.
[35] J.E. Fritts, W.H. Wolf, B. Liu, Understanding multimedia application characteristics for designing programmable media processors, in: Electronic Imaging'99, International Society for Optics and Photonics, 1998, pp. 2–13.
[36] S.-M. Moon, K. Ebcioğlu, A study on the number of memory ports in multiple instruction issue machines, in: Proceedings of the 26th Annual International Symposium on Microarchitecture, MICRO 26, IEEE Computer Society Press, Los Alamitos, CA, USA, 1993, pp. 49–59.
[37] M.S. Lam, R.P. Wilson, Limits of control flow on parallelism, in: Proceedings of the 19th Annual International Symposium on Computer Architecture, ISCA '92, 1992, pp. 46–57.
[38] T.M. Austin, G.S. Sohi, Dynamic dependency analysis of ordinary programs, SIGARCH Comput. Archit. News 20 (2) (1992) 342–351.
[39] B.A. Abderazek, M. Masuda, A. Canedo, K. Kuroda, Natural instruction level parallelism-aware compiler for high-performance queuecore processor architecture, J. Supercomput. 57 (3) (2011) 314–338.
[40] A. Ashouri, V. Zaccaria, S. Xydis, G. Palermo, C. Silvano, A framework for compiler level statistical analysis over customized VLIW architecture, in: IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC), 2013, pp. 124–129.
[41] V. Brost, F. Yang, C. Meunier, Flexible VLIW processor based on FPGA for efficient embedded real-time image processing, J. Real-Time Image Process. 9 (1) (2014) 47–59.
[42] D. Stevens, V. Chouliaras, V. Azorin-Peris, J. Zheng, A. Echiadis, S. Hu, BioThreads: a novel VLIW-based chip multiprocessor for accelerating biomedical image processing applications, IEEE Trans. Biomed. Circ. Syst. 6 (3) (2012) 257–268.
[43] J.A. Fisher, Trace scheduling: a technique for global microcode compaction, IEEE Trans. Comput. 30 (7) (1981) 478–490.
[44] M.D. Smith, M.S. Lam, M.A. Horowitz, Boosting Beyond Static Scheduling in a Superscalar Processor, vol. 18, ACM, 1990.
[45] M. Lam, Software pipelining: an effective scheduling technique for VLIW machines, ACM SIGPLAN Notices, vol. 23, ACM, 1988, pp. 318–328.
[46] W. Pugh, The omega test: a fast and practical integer programming algorithm for dependence analysis, in: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, ACM, 1991, pp. 4–13.
[47] D.E. Maydan, J.L. Hennessy, M.S. Lam, Efficient and exact data dependence analysis, ACM SIGPLAN Notices, vol. 26, ACM, 1991, pp. 1–14.
[48] R.P. Wilson, M.S. Lam, Efficient Context-sensitive Pointer Analysis for C Programs, vol. 30, ACM, 1995.
[49] C. McNairy, D. Soltis, Itanium 2 processor microarchitecture, IEEE Micro 23 (2) (2003) 44–55.
[50] S. Dutta, A. Wolfe, W. Wolf, K.J. O'Connor, Design issues for very-long-instruction-word vlsi video signal processors, in: Workshop on VLSI Signal Processing, IX, IEEE, 1996, pp. 95–104.
[51] V. Lapinskii, M. Jacome, G. de Veciana, Application-specific clustered VLIW datapaths: early exploration on a parameterized design space, IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. 21 (8) (2002) 889–903.
[52] Z. Wu, W. Wolf, Data-path synthesis of VLIW video signal processors, in: Proceedings. 11th International Symposium on System Synthesis, IEEE, 1998, pp. 96–101.
[53] D.W. Wall, Limits of Instruction-level Parallelism, vol. 19, ACM, 1991.
[54] N. Goel, A. Kumar, P.R. Panda, Shared-port register file architecture for low-energy VLIW processors, ACM Trans. Archit. Code Optim. 11 (1) (2014) 1:1–1:32.
[55] V.H. Allan, R.B. Jones, R.M. Lee, S.J. Allan, Software pipelining, ACM Comput. Surv. (CSUR) 27 (3) (1995) 367–432.