Application Specific Multi-port Memory Customization in FPGAs

Gorker Alp Malazgirt, Hasan Erdem Yantır and Arda Yurdakul
Computer Engineering
Bogazici University, Bebek, Istanbul
alp.malazgirt, hasanerdem.yantir, yurdakul@boun.edu.tr

Smail Niar
LAMIH
University of Valenciennes, Valenciennes, France
smail.niar@univ-valenciennes.fr

***Abstract*—FPGA block RAMs (BRAMs) offer speed advantages compared to LUT-based memory designs but a BRAM has only one read and one write port. Designers need to use multiple BRAMs in order to create multi-port memory structures which are more difficult than designing with LUT-based multiport memories. Multi-port memory designs increase overall performance but comes with area cost. In this paper, we present a fully automated methodology that tailors our multi-port memory from a given application. We present our performance improvements and area tradeoffs on state-of-the-art string matching algorithms.**

I. Introduction

In FPGAs, multi-port memories are extensively used in soft processors and custom hardware accelerators for increasing performance. Current mainstream soft processors in FPGAs are RISC processors but for extensive parallelism in FPGAs, soft VLIW processors are also suggested. Although VLIW processors allow more parallelism, they are difficult to optimize because of larger memory requirements than RISC processors. This is due to increasing port numbers, code size and register usage for supporting more parallelism. Performance improvement through exploiting parallelism has costs. Software designers need to parallelize sequential applications. Hardware designers need design exploration in order to utilize hardware resources and design hardware that suits the requirements of given applications.

In this paper, we present a fully automated methodology that tailors the multi-port memory from a given application for a generic VLIW processor that can be customized for a given application. This method leverages instruction level parallelism of memory and ALU operations from a sequential algorithm's execution trace. Given an application, we automatically extract parallelism from execution traces to determine its required input output port numbers that leverages parallelism. Then, we reorder the traces and bundle them and simulate the performance with the FPGA technology file that supplies the technology characteristics such as delay, area and power for an estimation of resource utilization and execution speed of the application.

II. Related Work

A. Extracting Instruction Level Parallelism (ILP)

Parallelism extraction and parallelizing instructions have been applied widely in compiler domain[1][2][3] [4]. Parallelizing compilers for VLIW processors provides ILP by encapsulating more than one operations in one instruction at compile time. Software pipelining, trace scheduling, predicated execution, hierarchical reduction and speculative execution have been major compiler optimizations. Trace scheduling [1] requires additional code when operations are reordered. Speculative execution [2] reduces the compensation code and moves instructions past branches. Software pipelining [3]aims at compacting loop kernels by minimizing initiation intervals. Predicated execution [4] aims to convert branches to basic blocks with hardware defined predicates on certain instructions. Hierarchical reduction [3] is the method to simplify rescheduling process by compacting and representing scheduled program components as a single component. Approaches that extract ILP parallelism without using execution traces require dependency analysis methods such as points-to-analysis [5] which is computationally expensive. Our method analyzes execution traces where all the address dependencies can be checked via real memory addresses. Register dependencies can be checked via register locations. Control Flow Graph (CFG) is extracted and used to determine loop bounds and inter loop dependencies.

B. Multi-port Memories in FPGAs

The most efficient method of designing multi-port memories in FPGAs is using block RAMs (BRAM) that are available in traditional FPGAs [6] [7]. However, a BRAM has only two ports, which are not sufficient to implement a highly parallel architecture that usually needs multiple ports. The third method is combining on-chip BRAMs in conventional methods to form a multi-port memory. In multi-port memory design with BRAMs, two common methods are replication and banking. Replication and banking can be exploited in order to increase the number of read and write ports [8][9]. However in this method, memory is partitioned between the banks. Multiple-read can be done from the same bank, but two or more write operations cannot be done on the same bank. In order to handle this problem, a set of software and hardware methods were proposed by Anjam et al [10][11]. In this work, we have used the multi-port memory design suggested by [9] because it has shown to provide better multi-port performance explained in the work.

III. Application Analysis and Parallelism Extraction

Execution traces are generated from the application binary and the input data set. Our trace generation program uses PIN [12] library and application binaries are compiled from C/C++ source files.

For parallelism analysis, we have derived rules for extracting parallelism from a sequential algorithm. These rules reorder and bundle instructions without breaking data dependencies. For preventing structural hazards, we assume that there are sufficient number of internal registers in the system and register renaming is employed to prevent structural hazards. The details of parallelism extraction are explained in our previous work [13].

Instructions that are bundled together are parallel and can be scheduled at the same access step. This reordering extracts the maximum parallelism which is defined as the highest number of instructions in an access step.

Direct implementation of a multi-port memory which can support maximum parallelism will be inefficient in terms of area and power consumption though it provides the fastest execution time. Our recommendation algorithm described in [13] reduces maximum port usage and issue slot size and keeps maximum parallelism. Hence, we gain significantly from resource usage while keeping the performance attained at maximum parallelism.

IV. Multi-port Memory Architecture

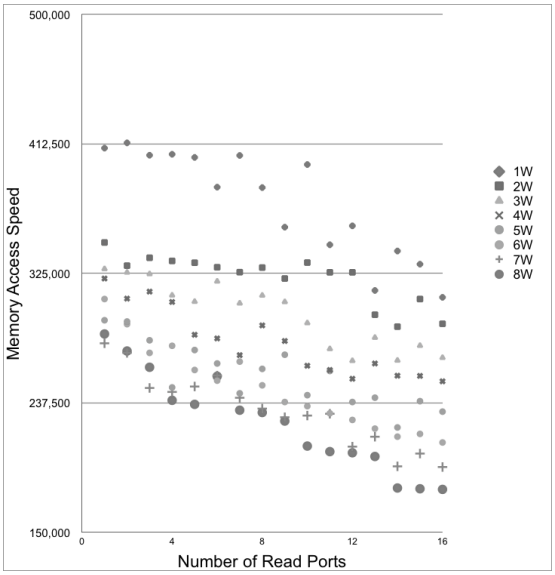
In traditional FPGA architectures, there exist dedicated BRAMs for storage [6] [7]. These BRAMs have two ports which are used either as write or read port depending on the BRAM mode. In true dual port mode, each port can change its read/write behavior during run time i.e. 2R or 1R & 1W or 2W. In simple dual port mode, a BRAM has exactly dedicated one read and one write port and configuration cannot be changed during run time.

We have synthesized 64x512 bit memory with minimum delay constraints and reported post-place and route maximum clock period. The minimum delay constraints are reduced iteratively by our automation tool. At each iteration, our tool lowers the minimum delay constraint and synthesizes the design. The process continues until the minimum delay constraint is not met with 0.01 ns difference.

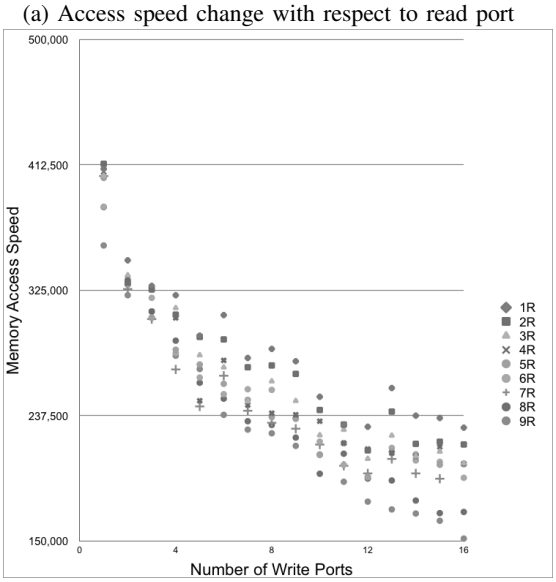
The results have been generated on Xilinx Virtex-5 XC5VLX110T and Zynq-7020. Figure 1a and Figure 1b show that increasing number of write ports decreases memory access speed more than increasing the number of read ports for Virtex-5. Thus, though increasing the number of ports might guarantee the required parallelism, it might degrade performance due to unexpected slowdown of memory accesses. Zynq-7020 has also shown similar behavior therefore we only present Virtex-5 results.

V. Experimental Results

We developed a custom simulator in order to compare the performance of our multi-port memory and available



(a) Access speed change with respect to read port



(b) Access speed change with respect to write port

Fig. 1: Memory access speed change with respect to read and write ports of 64x512 bit memory on Virtex-5

dual port memories on FPGAs. All multi-port and dual-port memories are configured and synthesized automatically by our custom tool using C# and Xilinx TCL scripting. Our tool also generates the FPGA technology file from synthesis results.

We apply our method on string matching algorithms shown in Table I. String matching has been used extensively and different approaches have been developed. Nevertheless, all of them are highly memory intensive [14]. Some of the algorithms we experiment have different variations. In FSBNDMQ, we change the q grams and the selected lookahead values. In FAOSO, alignment numbers vary. In TVSBS, we used different window sizes.

Input set contains one million characters of human DNA. In this text, we search for a pattern with a length of ten thousand

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

TABLE I: String Matching Algorithms and abbreviations

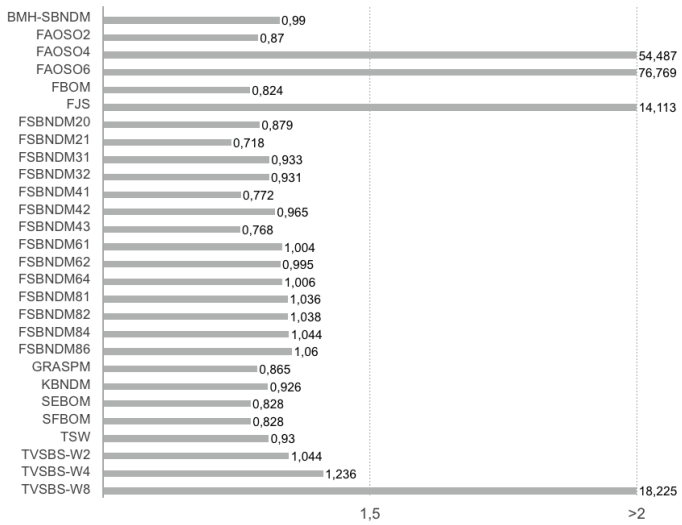


Fig. 2: Area-Delay product of multi-port memory configuration normalized to true dual port

characters and each character is one byte. Hence a pattern is 10kB long. The text alphabet size is four. Loading data from external memory to FPGA memory is handled by custom logic controller which controls a multi-paged memory architecture. Initially, all the pages are filled with data. The number of active pages and idle pages are determined according to the algorithms' behavior. Idle pages are loaded with new data. This mechanism allows memory references to take constant amount of cycles. In addition, we assume that the change of input data set doesn't affect the algorithm behavior. All our multi-port, true dual port and simple dual port memory configurations have 32 bits word size and depth is 4096. Each multi-port memory with 32*4096 bit configuration allows us to store a pattern, because this configuration can store 16KB of data. Xilinx Zynq-7020 has 560KB of memory [15], therefore our paged memory architecture can load up to $560KB/16KB = 35$ different pages.

Performance of the string matching algorithms of multi issue multi-port (MP) memory are compared with single issue true dual port (TDP) and single issue simple dual port configurations (SDP). The obtained execution times are represented as e_{MP}, e_{TDP} and e_{SDP} respectively in Table II. Operation costs for non-memory operations are taken from [16]. Issue size of non-memory operations are limited to the maximum number of read/write port number. For example, 4R 1W configuration is assumed to have less than or equal to five issue slots.

All of the algorithms have shown speed ups between 1.55x to 4.25x in MP over TDP and SDP memories shown in Table II. In addition, experiments show that algorithms with higher average parallelism do not necessarily yield better performance. FSBNDMQ algorithm has the least average parallelism and least ratio of memory instructions among all algorithms. FAOSO6 has the largest number of read/write ports and TVSBS-w8 has the highest average parallelism. However, they are both slower than FSBNDMQ. The average parallelism does not seem to correlate with access step values. The average parallelism of FJS algorithm is 4.13 whereas FSBNDMQ implementations have 2.85 on average. Nevertheless, FJS has six times more access steps than FSBNDMQ. Similarly, read/write port numbers do not convey a hint about the recommended access steps. GRASPM has lower number of read/write ports than FAOSO6, but it requires more access steps to complete.

The number of read and write ports affect FPGA utilization differently. SDP has the lowest resource usage. TDP employs more BRAMs than SDP. Depending on the number of read ports, MP configurations use more BRAMs than TDP and SDP. All configurations with single write port use only one LUT. MP with multiple write ports use more LUTs and BRAMs than equivalent single port configurations. For example, 3R 2W configuration consumes more BRAMs and LUTs than 4R 1W configuration.

Figure 2 presents the area-delay product of multi-port configurations which are normalized to true dual port configurations. Results show that majority of multi-port configurations are more efficient than dual port. However, algorithms with multiple write ports are inefficient because the usage of BRAM and LUT is very high compared to dual port memory configuration.

VI. Conclusion

In this work, we present a method which extracts instruction level parallelism from sequential algorithm and tailors our multi-port memory by recommending multi-port memory size and issue slot size. Our custom simulator perform performance estimations comparing our multi-port memory over true dual and simple dual port memory on FPGAs. Our method has improved FSBDMQ algorithm by 3x with Area-Delay product of 0.7 compared to true dual port BRAM memory.

Acknowledgment

This work is supported by Bogazici University Scientific Research Projects (BAP) Project No: 11A01P6 and by

Benchmark	Read & Write Number	Average Parallelism	FPGA Utilization (BRAM, LUT)	e_{MP} (μs)	e_{TDP} (μs)	e_{SDP} (μs)	e_{MP}/e_{TDP}	e_{MP}/e_{SDP}
FSBNDMQ20	3R, 1W	2.99	(24, 1)	7437	21156	24030	2.84	3.23
FSBNDMQ21	2R, 1W	2.82	(16, 1)	7576	21098	23962	2.78	3.16
FSBNDMQ31	3R, 1W	2.82	(24, 1)	7041	18865	21514	2.68	3.06
FSBNDMQ32	3R, 1W	2.79	(24, 1)	8474	22755	25880	2.69	3.05
FSBNDMQ41	2R, 1W	2.81	(16, 1)	7931	20542	23460	2.59	2.96
FSBNDMQ42	3R, 1W	2.80	(24, 1)	7956	20606	23530	2.59	2.96
FSBNDMQ43	2R, 1W	2.83	(16, 1)	8027	20908	23876	2.60	2.97
FSBNDMQ61	3R, 1W	2.84	(24, 1)	9707	24165	27681	2.49	2.85
FSBNDMQ62	3R, 1W	2.88	(24, 1)	9711	24388	27921	2.51	2.88
FSBNDMQ64	3R, 1W	2.84	(24, 1)	9830	24422	27968	2.48	2.85
FSBNDMQ81	3R, 1W	2.88	(24, 1)	11327	27334	31417	2.41	2.77
FSBNDMQ82	3R, 1W	2.87	(24, 1)	11322	27276	31349	2.41	2.77
FSBNDMQ84	3R, 1W	2.86	(24, 1)	11338	27158	31213	2.40	2.75
FSBNDMQ86	3R, 1W	2.85	(24, 1)	12728	30017	34540	2.36	2.71
BMH-SBDNM	4R, 1W	3.25	(31, 1)	14259	42306	48287	2.97	3.39
KBNDM	4R, 1W	3.64	(31, 1)	25947	82338	94942	3.17	3.66
FAOSO2	3R, 1W	3.30	(24, 1)	25837	74215	85624	2.87	3.31
FAOSO4	2R, 6W	3.29	(96, 113)	21801	47615	54827	2.18	2.51
FAOSO6	2R, 6W	4.1	(96, 113)	56569	87688	102240	1.55	1.81
SEBOM	3R, 1W	3.54	(24, 1)	37081	111952	129271	3.02	3.49
FBOM	3R, 1W	3.52	(24, 1)	37635	114203	131507	3.03	3.49
SFBOM	3R, 1W	3.48	(24, 1)	37709	113828	131126	3.02	3.48
TVSBS-w2	4R, 1W	3.30	(31, 1)	28681	80734	93695	2.81	3.27
TVSBS-w4	5R, 1W	3.39	(40, 1)	32468	91970	106828	2.83	3.29
TVSBS-w8	3R, 2W	4.16	(48, 49)	109153	311446	353495	2.85	3.24
TSW	4R, 1W	3.26	(31, 1)	24067	76013	87815	3.16	3.65
FJS	3R, 2W	4.13	(48, 49)	45628	114203	193738	3.68	4.25
GRASPM	3R, 1W	3.70	(24, 1)	25907	74901	86501	2.89	3.34

TABLE II: R & W port sizes, average parallelism, FPGA resource utilizations and simulation time of string matching algorithms

TUBITAK given to Prof. S. NIAR under Project No: 2221.

REFERENCES

[1] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, pp. 478–490, 1981.

[2] M. D. Smith, M. S. Lam, and M. A. Horowitz, *Boosting beyond static scheduling in a superscalar processor*. ACM, 1990, vol. 18, no. 3a.

[3] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," in *ACM Sigplan Notices*, vol. 23, no. 7. ACM, 1988, pp. 318–328.

[4] P. Hsu and E. S. Davidson, *Highly concurrent scalar processing*. IEEE Computer Society Press, 1986, vol. 14, no. 2.

[5] R. P. Wilson and M. S. Lam, *Efficient context-sensitive pointer analysis for C programs*. ACM, 1995, vol. 30, no. 6.

[6] Xilinx, *IP Processor Block RAM (BRAM) Block*.

[7] Altera, *Internal Memory (RAM and ROM) User Guide*.

[8] M. A. R. Saghir and R. Naous, "A configurable multi-ported register file architecture for soft processor cores," in *Proceedings of the 3rd international conference on Reconfigurable computing: architectures, tools and applications*, ser. ARC'07, 2007, pp. 14–25.

[9] H. Yantir, S. Bayar, and A. Yurdakul, "Efficient implementations of multi-pumped multi-port register files in fpgas," in *Digital System Design (DSD), 2013 Euromicro Conference on*, Sept 2013, pp. 185–192.

[10] F. Anjam, S. Wong, and F. Nadeem, "A multiported register file with register renaming for configurable softcore vliw processors," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec., pp. 403–408.

[11] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '10, 2010, pp. 41–50.

[12] C.-K. Luk and Cohn, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005.

[13] G. Malazgirt, A. Yurdakul, and S. Niar, "Mipt: Rapid exploration and evaluation for migrating sequential algorithms to multiprocessing systems with multi-port memories," in *High Performance Computing and Simulation (HPCS), 2014 International Conference on*, July 2014.

[14] S. Faro and T. Lecroq, "The exact online string matching problem: A review of the most recent results," *ACM Comput. Surv.*, vol. 45, no. 2, pp. 13:1–13:42, Mar. 2013.

[15] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*.

[16] Intel, *Intel Software Developer Manuals*. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html