Parallel Memory Architecture for TTA Processor

Jarno K. Tanskanen[1], Teemu Pitkänen[1], Risto Mäkinen[2], and Jarmo Takala[1]

[1] Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland
jarno.tanskanen@tut.fi, teemu.pitkanen@tut.fi, jarmo.takala@tut.fi
[2] Plenware Oy, P.O. Box 13, FIN-33201 Tampere, Finland
risto.makinen@plenware.fi

Abstract. A conflict resolving parallel data memory system for Transport Triggered Architecture (TTA) is described. The architecture is generic and reusable to support various application specific designs. With parallel memory, more area and power consuming multi-port memory can be replaced with single-port memory modules. Number of ports can be increased over what is available on a design library for multi-port memories. In an FFT TTA example, dual-port data memory was replaced by the proposed architecture. To avoid memory conflicts, the original code was rescheduled and the TTA core was regenerated for the new schedule. The original memory required an area higher by a factor of 3.38 and energy higher by a factor of 1.70. In this case, the energy consumption of the processor core increased so that system energy consumption remained about the same. However, the original system required an area higher by a factor of 1.89.

1 Introduction

TTA [1] belongs to a class of statically scheduled processors exploiting instruction level parallelism and resembles VLIW architecture. TTA framework can be efficiently used to generate optimized application specific cores, e.g., to DSP, telecommunication, and multimedia fields. Several applications from these fields contain well exploitable parallelism which can be used by increasing processing resources. As a result, higher performance can be obtained or power consumption could be decreased if the processing requirements are met with a lower clock frequency. Increasing the processing resources leads often also to the higher data bandwidth need which can be provided by multiple data memory ports. Multi-port memories have higher power consumption, require larger area, and longer access time than equally sized single-port memories. To avoid the cost of actual multi-port memory structure, the following different methods have been used in multiple-issue processors [2,3,4]. To provide n-port functionality, n single port memory modules with the same data content could be employed. A write operation is always sent to all the memory modules to maintain the data coherence. As a drawback, the memory must be replicated n times and no other accesses can be made during a write operation. A single port memory can be also accessed with a higher frequency than the processing frequency. However, this solution might not scale to higher port numbers. Finally, n single port memories having total size of multi-port memory can be used to emulate the multi-port memory. Additional permutation and address computation circuitry is needed. Also, memory conflicts may exist. This is referred as parallel memory architecture. More recent multi-port memories have been presented in

S. Vassiliadis et al. (Eds.): SAMOS 2007, LNCS 4599, pp. 273–282, 2007.

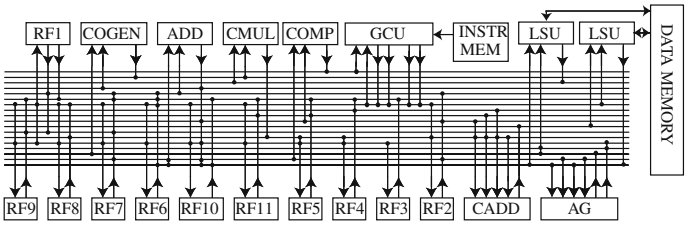

Fig. 1. Architecture of the FFTTA processor. ADD: Real adder. AG: Data address generator. CADD: Complex adder. CMUL: Complex multiplier. COGEN: Coefficient generator. COMP: Comparator unit. GCU: Global control unit. LSU: Load-store unit. RFx: Register files, containing total of 23 general purpose registers.

[5,6,7,8]. The parallel memory approach is the most promising to us since the application specific designs may have regular and predictable memory access patterns and the existing parallel memory theory can be used to construct specific storage schemes to avoid or significantly reduce the memory conflicts. Furthermore, the permutation and address computation circuitry might be fitted in the existing pipeline structure of the processor without lowering the clock frequency or increasing the number of pipeline stages. Often, the designs which consider conflict resolving multi-port memory architecture, employ some form of a simple low-order interleaving scheme [2,4,6,8]. On the other hand, new storage scheme proposals, like the one in [9] employed in this paper, concentrate to conflict-free, complex memory storage and rarely consider conflict resolving support. This paper presents a hardware in detail for dynamic conflict resolving. The proposed parallel memory architecture supports also alternative storage schemes.

2 TTA Processor Architecture

In the TTA programming model, the program specifies only the data transports to be performed by the interconnection network and operations occur as "side-effect" of data transports [1]. Operands to a function unit are input through ports and one of the ports is dedicated to be a trigger. When data is moved to the trigger port, execution of an operation is initiated. A TTA processor consists of a set of function units and register files of general-purpose registers. These structures are connected to an interconnection network, which connects the input and output ports of the resources. The architecture can be modified by adding or removing resources. Furthermore, special function units with user-defined functionality can be easily included. The structural VHDL description of the TTA core can be obtained using the processor generator of the TCE framework [10]. As an example, the FFTTA core [11,12] employed in Sec. 6 is illustrated in Fig. 1.

3 Parallel Memory

In a parallel memory system, a module assignment function $S(i)$ is a function of the incoming address i from the LSU and determines the index of the memory module MM

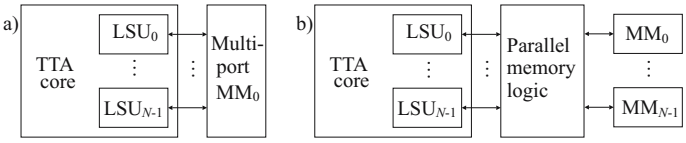

Fig. 2. Different data memory configurations: a) multi-port memory and b) single-port memories with parallel memory logic

where the data is located. The address for $\mathrm{MM}_{S(i)}$ is determined by the address assignment function $a(i)$. If the parallel memory logic has N LSUs, then N module $S(i_k)$ and address $a(i_k)$ assignment functions are computed simultaneously, when i_k refers to an address from LSU_k, $0 \leq k < N$. Basically, $S(i)$ determines, how well the memory performs for a given parallel address trace. The most simple and well-known module assignment functions are low-order and high-order interleaving functions. Low-order interleaving, $S(i) = i \bmod N$, $a(i) = i/N$, is efficient for parallel access of successive array elements. High-order interleaving, $S(i) = i/(a_{max}+1)$, $a(i) = i \bmod (a_{max}+1)$, performs well when several different arrays are accessed in parallel. The term $(a_{max}+1)$ is simply a constant telling the number of locations in each MM. In many cases, the operands for parallel processing can be stored so that conflict-free access to certain patterns is possible, e.g., rows, columns, blocks, forward- and backward-diagonals [13,14]. These are called the access formats. In general, the module assignment functions used for this purpose are linear [13] or so called XOR-schemes [15,16]. Multi-skewing scheme [17] provides versatile access formats. Storage schemes supporting stride accesses are presented in [18,19]. A brief overview of storage schemes is provided in [20]. For an FFT we employ a generic FFT parallel memory scheme proposed in [9].

4 Parallel Memory in a System

As shown in Fig. 2, the parallel memory logic is designed to locate between the load-store units (LSUs) and synchronous single-port memory modules (MMs). Parallel memory non-optimally emulates multi-port memory. Unlike in multi-port memories, in parallel memories there can be memory conflicts, i.e., one or more single port MMs are tried to be accessed more than once during a single cycle parallel memory access. It is not possible to find a generic storage scheme that is conflict-free for all address traces. In the case of memory conflict, parallel memory hardware recognizes the conflict, locks the processor by sending a lock request to the global control unit (GCU) of the TTA, performs conflicting accesses sequentially (requiring more cycles), and releases the lock. No modifications to software are required for correct functionality. Because of this locking behavior, the software does not know whether multi-port or parallel memory system is employed. However, possible conflicts increase the cycle count.

A multi-port memory can be replaced by a parallel memory architecture after the application code has been written. In this case, the software has been likely developed assuming an ideal multi-port memory and no attention is paid into the addresses of

simultaneous memory accesses. It can be possible not to find a conflict-free parallel memory storage scheme since the address traces can be irregular, not fitting to typical access formats. A better performance in terms of clock cycles can be obtained when a conflict-free storage scheme is found for application specific access formats which are used in the application code. Especially, the code of the innermost loops, where typically the most of the execution time is spent, should be written so that the addresses of simultaneous memory accesses would not cause memory module access conflicts. This may require manual assembler optimization and regeneration of the hardware for the TTA core. However, as will be seen in Sec. 6, care must be taken since the modifications can increase the power consumption of the core.

The proposed parallel memory design is generic and re-usable. Application specific memory functions can be fitted in and generics are employed in the VHDL design so that several parallel memory components with different parameters (buswidths, number of ports, and memory functions) could be fitted in the same design. Parallel memory logic provides the needed address computation, interconnection, and conflict resolving logic. This is a matched memory system, i.e., the number of load store units (LSUs) and memory modules (MMs) is the same N. Often power-of-two N is preferred.

5 Conflict Resolving Parallel Memory Architecture

Depending on the address i_k and the module assignment function $S(i_k)$, the load or store operation from LSU_k may refer to any MM. For this reason, crossbars are needed to route the MM input signals from the LSUs to the MMs. A crossbar is also needed to route the read (load) data from the MMs back to the LSUs, which made the corresponding load requests. These crossbars, related to read and write operations, are illustrated in Fig. 3 for $N = 4$. The read data from the MM needs to be saved to Rlatch registers at the correct moment when several accesses are made to the same MM during the conflict resolving. Rlatches are controlled by simple state machines.

For each LSU_k and MM_k pair, there is a control unit which correctly enables the MM_k and drives the crossbar mux controls for the address ($\mathrm{AddrMuxCtrl}_k$), read data ($\mathrm{RdMuxCtrl}_k$), and write data ($\mathrm{WrMuxCtrl}_k$) crossbar muxes related to MM_k. This control unit circuitry is shown in Fig. 5. The module assignment functions $S(i_k)$, $0 \leq k < N$ are solved in parallel. They can be used to control read data crossbar so that $S(i_k)$ drives $\mathrm{RdMuxCtrl}_k$ (connected to RdMux_k in Fig. 3) at the correct moment. For various other control purposes, $S(i_k)$ indices are binary decoded and used to construct a binary control matrix. This is shown in Fig. 5, where a decoder produces a single control matrix row, $\mathrm{CtrlMatrixRow}_k$. An example matrix is shown in Fig. 4a for $N = 4$. It can be seen that LSU_0 and LSU_3 are accessing MM_3, and LSU_1 and LSU_2 are accessing MM_2.

Each control unit needs to know which LSUs will access their memory module. This is obtained by transposing the control matrix. A transposed control matrix is shown in Fig. 4b. The kth row of the transposed matrix is delivered for the kth control unit. In Fig. 5, the control matrix composed from CtrlMatrixRows from all the control units is transposed and $\mathrm{TCtrlMatrixRow}_k$ is obtained for each control unit k. If there are more than one '1' bits on any TCtrlMatrixRow of the transposed matrix, then there are corresponding number of memory conflicts. Parallel control units make always as

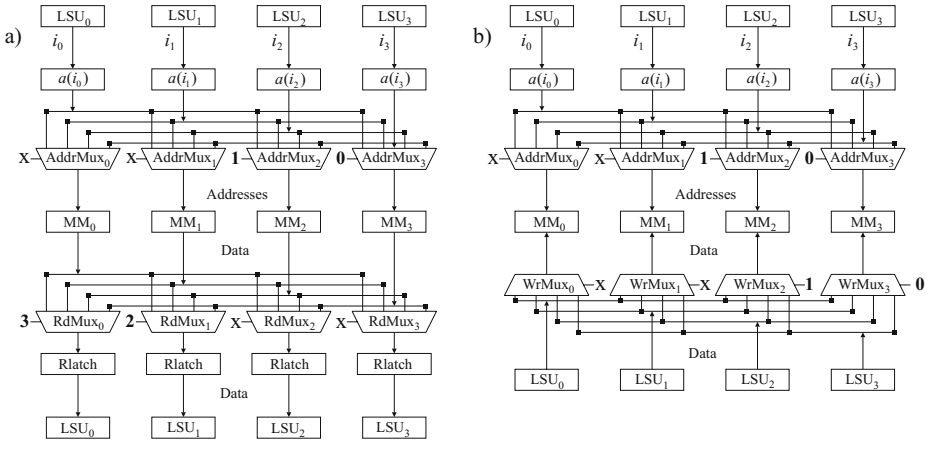

Fig. 3. Parallel data memory a) read (load) and b) write (store) operation examples and related address, read data, and write data crossbars. The fixed control signals for the muxes refer to the example case from the next page.

a) Control matrix.

$\mathbf{MM}_k$	0	1	2	3	
LSU_0	0	0	0	1	$S(i_0)$
LSU_1	0	0	1	0	$S(i_1)$
LSU_2	0	0	1	0	$S(i_2)$
LSU_3	0	0	0	1	$S(i_3)$

b) Transposed control matrix.

$\mathbf{LSU}_k$	0	1	2	3
MM_0	0	0	0	0
MM_1	0	0	0	0
MM_2	0	1	1	0
MM_3	1	0	0	1

c) '1' bits served in the first cycle.

$\mathbf{LSU}_k$	0	1	2	3
MM_0	0	0	0	0
MM_1	0	0	0	0
MM_2	0	0	1	0
MM_3	0	0	0	1

d) '1' bits served in the second cycle.

$\mathbf{LSU}_k$	0	1	2	3
MM_0	0	0	0	0
MM_1	0	0	0	0
MM_2	0	1	0	0
MM_3	1	0	0	0

Fig. 4. An example control matrix

many parallel accesses as possible. If there is a conflict, a priority encoder (PriEncoder) of the control unit k selects the rightmost bit (LSU) on the TCtrlMatrixRow_k in the first cycle, the next rightmost bit in the second cycle, and so on, until all the '1' bits on the row are served. '1' bits are reduced one by one with the XOR-ports producing $\text{NewTCtrlMatrixRow}_k$ in Fig. 5. This loop is enabled by the multiplexer controlled by LockrqReg signal. All the rows of the transposed control matrix are processed in parallel by the control units.

Each priority encoder has MoreToCome signal which tells if there are more '1' bits left. As is shown in Fig. 5, these MoreToCome signals are combined from each control unit to a single bit using an OR-tree. After registering, this bit becomes the lock request signal (LockrqReg) to be send to the processor. CurrLsu signal of the control unit tells the index of the LSU which is to be currently served by the memory module. The same position CurrLsu signal bits have been combined by N OR-trees shown in Fig. 5. The resulted CurrLsuEnBits are used to control the state machines and RdMuxCtrl signals.

The transposed control matrices for the first and second memory cycles of the example are shown in Figs. 4c and 4d, respectively. The rows of these matrices are used

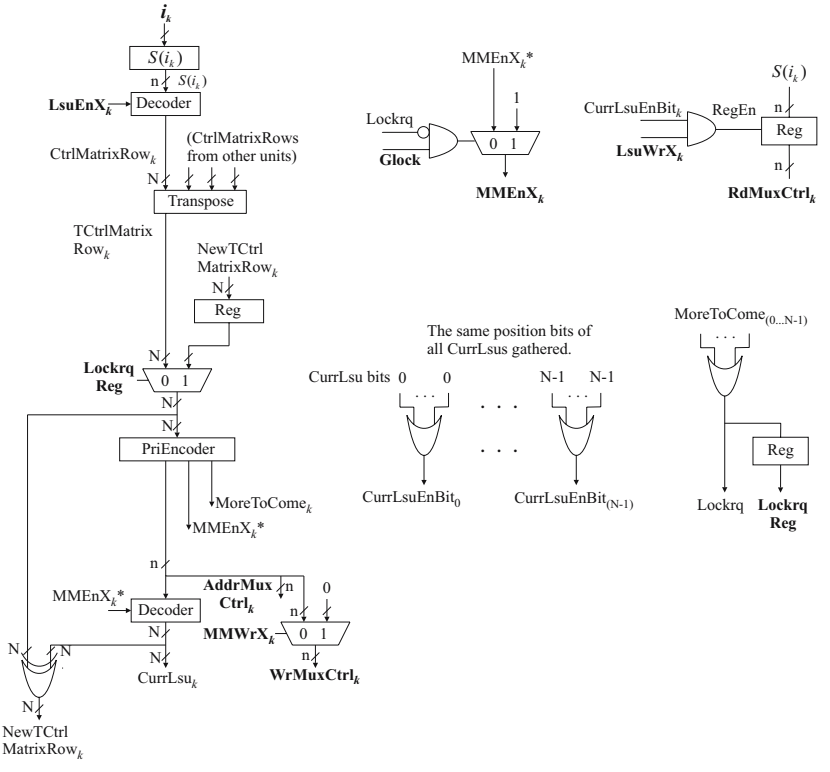

Fig. 5. Control unit logic. There is one control unit circuit for each LSU_k and MM_k pair. OR-trees producing CurrLsuEnBit and Lockrq signals combine the data from all the control units. N refers to the number of memory ports and n is the bit width of N. Block input and output signals are written in bold. LsuEnX_k, LsuWrX_k, and i_k are coming from LSU_k. MMEnX_k and MMWrX_k are enable and R/W signals for MM_k. (With true multi-port memory, we would simply have LsuEnX_k = MMEnX_k and LsuWrX_k = MMWrX_k.) Glock refers to the global lock signal coming from the processor core.

to control the corresponding muxes of the address (AddrMuxCtrl), write data (WrMuxCtrl), (write mask), and write signal crossbars. The crossbar mux control signals for the control matrix given in Fig. 4 would be the following (x refers to don't care condition):

When all the accesses are load operations, the mux controls are:

Cycle 1: $\mathrm{AddrMuxCtrl}_{0\ldots3} = \{x,x,2,3\}$, $\mathrm{RdMuxCtrl}_{0\ldots3} = \{x,x,2,3\}$.

Cycle 2: $\mathrm{AddrMuxCtrl}_{0\ldots3} = \{x,x,1,0\}$, $\mathrm{RdMuxCtrl}_{0\ldots3} = \{3,2,x,x\}$.

When all the accesses are store operations, the mux controls are:

Cycle 1: $\mathrm{AddrMuxCtrl}_{0\ldots3} = \mathrm{WrMuxCtrl}_{0\ldots3} = \{x,x,2,3\}$.

Cycle 2: $\mathrm{AddrMuxCtrl}_{0\ldots3} = \mathrm{WrMuxCtrl}_{0\ldots3} = \{x,x,1,0\}$.

The cases for cycle 2 are shown with a simplified architecture in Fig. 3. In practice, of course, a parallel memory access may consist of both load and store operations. The

main benefit of using the control matrix is that any kind of module assignment function, $S(i)$, can be included and used to construct the rows of the control matrix. After that, all the memory module and crossbar controls can be obtained automatically.

The lock request signal (LockrqReg) from the parallel memory logic is registered, and thus, the previous input values have to be saved to registers (i.e., addresses (i_k), memory enables (LsuEnX_k), write enables (LsuWrX_k), write data, and possible write masks), because otherwise they would be overwritten by new values. A simple state machine controls the saving of this data at the correct moment. The saving is not shown in the figures. The previous input values caused memory conflict(s) and the lock request and thus, they are needed to resolve the conflict(s) sequentially.

5.1 Scalable Hardware Modules

The design contains numerous scalable components including crossbars, OR-trees, decoders, and priority encoders:

- Three crossbars, one for addresses $a(i_k)$, read data, and write data. (When LSUs with subword support are used, an additional crossbar is needed for the write mask.) Each crossbar has N input and output busses with corresponding bus widths. In addition, one smaller crossbar for single-bit memory write signals is needed.
- $(N + 1)$ OR-trees (each OR-tree merges an N-bit input into a single bit).
- N priority encoders (each with an N-bit input).
- $2N$ decoders (each with a $log_2 N$-bit input).

When the number of ports N is increased, crossbars become more and more expensive, especially in terms of power and area, but also in delay. OR-trees, priority encoders, and decoders mainly affect the delay of the critical path of the design. The cost of the crossbars can be reduced by replacing one large parallel memory design with a couple of smaller designs. In that case, the LSUs would have access only to the MMs connected to their own parallel memory.

5.2 Pipelining

No new stages are added in the original system. The number of the LSU unit pipeline stages stays in three with the parallel memory architecture: the LSU signals going to and coming from the MMs are registered requiring two clock cycles, and the memory access itself requires one clock cycle. The input signals for the synchronous MMs (i.e., address, enable, write enable, write data (and write mask)) are read in the rising clock edge. Thus, because there is not much logic between the registered LSUs and input/output ports of the MMs, there is time available in the timing budget. The parallel memory logic between the LSU input and output registers (not shown) for read and write operations are illustrated in Fig. 6. The control logic, address crossbar, and write crossbar are located in the 1st cycle slot between the LSUs and MMs. The read crossbar is connected between the outputs of the MMs and LSUs. During a read clock cycle, the data appears to the MM data output bus after a related memory access delay. After that, the data goes through the read crossbar to the data input registers of the LSUs.

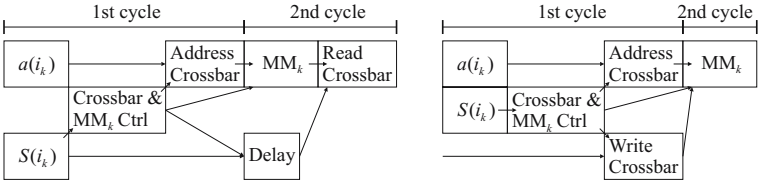

Fig. 6. Parallel memory pipelines: read on the left and write on the right

6 Experiments

In [11,12], a TTA processor for radix-4 1024-point FFT using a dual-port data memory is presented. To test the proposed parallel memory, the multi-port 2048×32 bits data memory was replaced with the parallel memory logic and two 1024×32-bit single-port MMs. A general form of the used storage scheme for FFT processors was presented in [9]. In our case $N = 2$ and the scheme reduces to a parity bit computation of an address i_k from the LSU_k. The computed bit $S(i_k)$ defines which one of the MMs should be accessed. An address for the module $S(i_k)$ is simply defined by $a(i_k) = i_k/N$. Due to memory access conflicts, the execution time was much longer with the FFT parallel memory scheme (7775 vs. 5234 cycles). (The low- and high-order schemes required 7775 and 9293 cycles, respectively.) The FFT memory scheme would be conflict-free if the parallel accesses would always consist of two subsequent input operand loads or alternatively, two subsequent result stores. However, in the original, manually written code, the first access of the parallel memory access always loads an input operand and the second access stores a result. Because of the in-place implementation, the result address trace starts to follow the input operand address trace after 10 clock cycles (excluding the first stage). This disturbs the regularity of the parallel address trace.

For conflict-free data accesses, the original code was now manually rescheduled and the hardware was reconfigured. The FFTTA was synthesized to a 130nm, 1.5V CMOS standard cell ASIC technology with Synopsys Design Compiler. This was followed by a gate level simulation at 250 MHz. Synopsys Power Compiler was used for the power analysis.

Table 1 provides various data for the considered implementations. The proposed double load/double store approach (C,D in Table 1) does not fit to the system so well as the original code (A,B), because now there is parallel operand access but not parallel function units to directly consume the operands (or produce the results). As a result, the core for the new schedule (C,D) required additional resources in terms of busses, registers, and register ports. The power consumption of the core increased significantly and the total energy consumption of the system remained about the same (A vs. D). However, the original system (A) required an area higher by a factor of 1.89 than the system with parallel memory (D). As expected, the data memory results were improved and the original data memory (A) required an area higher by a factor of 3.38 and an energy higher by a factor of 1.70 than the parallel memory (D).

The synthesis results for the parallel memory architecture only, with $N = 2, 4, 8$ ports (or MMs) are shown in Table 2. The size of a single-port MM was kept constant in 1024×32 bits. The related $S(i)$ and $a(i)$ were derived from the FFT storage scheme in

Table 1. Radix-4 1024-point FFT implementation on TTA

			Area/kgates	Energy/μJ	Power/mW	Clock cycles
A	Original Schedule	DP Data Mem	102.4	0.56	27.0	
		Core + others	37.6	0.98	46.6	
		Total	140.0	1.54	73.6	5234
B	Original Schedule	Par Data Mem	30.2	0.36	11.7	
		Core + others	37.7	1.01	32.5	
		Total	67.9	1.37	44.2	7775
C	Modified Schedule	DP Data Mem	102.4	0.57	27.2	
		Core + others	43.2	1.20	57.7	
		Total	145.6	1.77	84.9	5208
D	Modified Schedule	Par Data Mem	30.3	0.33	16.1	
		Core + others	43.7	1.21	58.2	
		Total	74.0	1.55	74.3	5208

Table 2. Demonstration of the scalability of the parallel memory implementation

		$N = 2$	$N = 4$	$N = 8$
Total memory size/32-bit words		2048	4096	8192
Area/kgates	Control logic	0.2	0.8	2.9
	Crossbars	1.9	6.1	16.7
	Memory	27.8	55.7	111.4
	Total	29.9	62.6	131.0
Clock period/ns		4.0	5.1	6.3

[9]. Note that the dual-port memory of size 2048×32 in Table 1 requires an area higher by a factor of 1.64 than the parallel memory of size 4096×32 with four ports.

7 Conclusion

A conflict resolving parallel data memory for TTA was proposed. With a parallel memory, more area and power consuming multi-port memory module can be replaced with single-port memory modules. For an application specific processor the address trace can be highly regular and predictable, and there can be a good change to find well performing storage scheme. The existing parallel memory theory can be used to construct application specific storage schemes. In an FFT TTA example, a dual-port data memory was replaced by the proposed architecture. To avoid memory conflicts, the original code was rescheduled and the TTA core was regenerated for the new schedule. Care must be taken in this approach since, e.g., in this specific case, the power consumption of the core increased enough to consume the power savings from the data memory. As a benefit, the area was divided by 1.89 compared to the original system.

References

1. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. John Wiley & Sons, Chichester, UK (1997)
2. Sohi, G.S., Franklin, M.: High-bandwidth data memory systems for superscalar processors. In: Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, U.S.A., pp. 53–62 (April 8-11, 1991)
3. Juan, T., Navarro, J.J., Temam, O.: Data caches for superscalar processors. In: Proc. 11th Int. Conf. Supercomputing, Vienna, Austria, pp. 60–67 (July 7-11, 1997)
4. Rivers, J.A., Tyson, G.S., Davidson, E.S., Austin, T.M.: On high-bandwidth data cache design for multi-issue processors. In: Proc. 30th Ann. ACM/IEEE Int. Symp. Microarchitecture, pp. 46–56. Research Triangle Park, NC, U.S.A (December 1-3, 1997)
5. Sawyer, N., Defossez, M.: Quad-port memories in Virtex devices. Xilinx application note, XAPP228 (v1.0) (September 24, 2002)
6. Zhu, Z., Johguchi, K., Mattausch, H.J., Koide, T., Hirakawa, T., Hironaka, T.: A novel hierarchical multi-port cache. In: Proc. 29th European Solid-State Circuits Conf., Estoril, Portugal, pp. 405–408 (September 16-18, 2003)
7. Patel, K., Macii, E., Poncino, M.: Energy-performance tradeoffs for the shared memory in multi-processor systems-on-chip. In: Proc. IEEE Int. Symp. Circuits and Systems, Vancouver, British Columbia, Canada, May 23-26, 2004, vol. 2, pp. 361–364. IEEE Computer Society Press, Los Alamitos (2004)
8. Ang, S.S., Constantinides, G., Cheung, P., Luk, W.: A flexible multi-port caching scheme for reconfigurable platforms. In: Bertels, K., Cardoso, J.M.P., Vassiliadis, S. (eds.) ARC 2006. LNCS, vol. 3985, pp. 205–216. Springer, Heidelberg (2006)
9. Takala, J.H., Järvinen, T.S., Sorokin, H.T.: Conflict-free parallel memory access scheme for FFT processors. In: Proc. IEEE Int. Symp. Circuits and Systems, Bangkok, Thailand, May 25-28, 2003, vol. 4, pp. 524–527. IEEE Computer Society Press, Los Alamitos (2003)
10. Jääskeläinen, P., Guzma, V., Cilio, A., Takala, J.: Codesign toolset for application-specific instruction-set processors. In: Proc. SPIE - Multimedia on Mobile Devices (2007)
11. Mäkinen, R.: Fast Fourier transform on transport triggered architectures. M.Sc. Thesis, Tampere University of Technology, Tampere, Finland (October 2005)
12. Pitkänen, T., Mäkinen, R., Heikkinen, J., Partanen, T., Takala, J.: Low-power, high-performance TTA processor for 1024-point Fast Fourier transform. In: Vassiliadis, S., Wong, S., Hämäläinen, T.D. (eds.) SAMOS 2006. LNCS, vol. 4017, pp. 227–236. Springer, Heidelberg (2006)
13. Budnik, P., Kuck, D.J.: The organization and use of parallel memories. IEEE Trans. Comput. C-20(12), 1566–1569 (1971)
14. Kim, K., Prasanna, V.K.: Latin squares for parallel array access. IEEE Trans. Parallel and Distrib. Syst. 4(4), 361–370 (1993)
15. Frailong, J.M., Jalby, W., Lenfant, J.: XOR-schemes: a flexible data organization in parallel memories. In: Proc. Int. Conf. Parallel Processing, pp. 276–283 (August 20-23, 1985)
16. Liu, Z., Li, X.: XOR storage schemes for frequently used data patterns. Journal of Parallel and Distributed Computing 25(2), 162–173 (1995)
17. Deb, A.: Multiskewing – a novel technique for optimal parallel memory access. IEEE Trans. Parallel and Distrib. Syst. 7(6), 595–604 (1996)
18. Rau, B.R.: Pseudo-randomly interleaved memory. In: Proc. 18th Ann. Int. Symp. Computer Architecture, Toronto, Ontario, Canada, pp. 74–83 (May 27-30, 1991)
19. Seznec, A., Lenfant, J.: Odd memory systems: a new approach. Journal of Parallel and Distributed Computing 26(2), 248–256 (1995)
20. Tanskanen, J.K., Creutzburg, R., Niittylahti, J.T.: On design of parallel memory access schemes for video coding. J. VLSI Signal Processing 40(2), 215–237 (2005)