A distributed cache memory system for custom vector processors*

João M. Meixedo and José C. Alves
{*jmeixedo@inescporto.pt, jca@fe.up.pt*}
FEUP / INESC-Porto

Abstract

This paper presents a parameterized distributed cache memory system for application specific processors implemented in FPGA devices. The system is made of several direct mapped cache memory modules that share the access to a single external data memory, and provide parallel data lanes that will feed the inputs of an arithmetic datapath. Each cache block is assigned to one or more application data vectors and includes a module to compute the effective memory address of each data value (32 bit), based on a reduced set of 4-bit commands that specify the iterations over up to 3 vector indexes. A prototype memory system was implemented and verified on a Virtex4LX80-10 FPGA, supporting one cycle reading latency of data located in the cache memory and a clock frequency of 200 MHz.

1. Introduction

Application specific vector processors can be an effective mean to improve the performance of conventional (scalar) processors. This is particularly interesting for embedded applications implemented in field-reconfigurable devices with integrated processors, where important gains in speed can be leveraged by custom designed deep pipelined datapaths to handle sequences of computations on vectors of data. Current FPGA devices can effectively host pipelines with tens of floating-point arithmetic operators, reaching performances up to a few giga flops. However, feeding the required data to minimize (ideally avoid) pipeline stalls can be impossible without the support for an adequate bandwidth to the data memory. This is the usual situation in FPGA-based systems where the main data memory is implemented by low cost dynamic memories that exhibit long reading latencies.

Vector architectures implementing the SIMD paradigm are being used for years to execute efficiently computing applications that perform operations on vectors of data. A vector processor extends the datapath of a conventional scalar CPU by including additional memories that form a vector register file, along with vector instructions that apply to the whole set of elements of the vector operands. Important performance gains can be achieved by building complex vector instructions that push their operands (vectors) through a pipelined datapath built by chaining arithmetic operators, as can be commonly identified in various sections of an application.

With current FPGAs it is possible to create deep pipelines with several floating point arithmetic operators and input operands. In spite of the high performance potential attained by such pipelines, to effectively use them it is necessary a convenient memory organization that may be able to provide enough data bandwidth to the datapath inputs. The ideal (and obvious) solution is to use dedicated memory banks to implement independent register files. However, limitations on the quantity of inter-chip memory available and the practical impossibility of populating discrete memories off-chip makes this approach usable only when the number and size of vectors used by an application is compatible with the quantity of memory that may be allocated to the vector registers.

In this paper we propose a parameterized and distributed cache memory system aimed to be implemented within a FPGA device, including dedicated but simple address generators for vector applications. The rest of the paper is organized as follows. Section 2 summarizes works of other authors related to the main subject of this paper. In section 3 a general overview of the memory system is presented. Sections 4 and 5 describe the architecture of the parameterized cache and the vector address generator associated with each cache block, respectively. Finally, section 6 summarizes the preliminary results and concludes the paper presenting plans for future developments.

2. Related work

Vector processing is being used for several years in high end processors and supercomputers to effectively exploit the data-level parallelism observed in many computing applications [1]. Until the appearance of high-density FPGAs by the late 90's, vector processing was an exclusive feature of commercial high-performance processors, application oriented processors like DSPs or GPUs or sophisticated custom designed machines.

Current FPGAs that include several inter-chip arithmetic functions and memory blocks offer now a technology capable of supporting practical applications of custom vector processing as a mean to meet the performance requirements of demanding embedded applications. This has motivated the development of vector processing units for embedded applications that act as auxiliary processors of conventional CPUs. Making use of hardware customization, the specific needs of a problem (eg. num-

*This work is funded by FCT (*Fundação para a Ciência e Tecnologia*), project PTDC/EEA-ELC/71556/2006

ber of processing lanes or organization of vector register file) can be exploited to better utilize the limited hardware resources of FPGA devices. Customizable and scalable vector FPGA-based co-processors were proposed in recent works [2, 3, 4], as a means to increase the computing power of embedded systems based on on-chip soft processors, like the MicroBlaze or the NIOS-II.

Targeting CMOS (non-configurable) technology, the VIRAM architecture [5, 6] developed at the University of California at Berkeley, USA, is a scalable vector co-processor for the 64-bit MIPS core that implements a multi lane processing core with a centralized vector register file, aimed for multimedia applications. A different microarchitecture from the same authors CODE [7] introduces a clustered vector register file that distributes the vector registers defined in the ISA by different (physical) groups, thus reducing the data traffic among functional units.

Memory access bandwidth is a key issue that affects significantly the performance of vector processors. The gains in speed obtained by processing vectors of data can only be effective when the memory system is capable of providing the required operands to the arithmetic units as close as possible to the fastest rate allowed by the datapath, thus avoiding pipeline stalls. Because it is not practical, mainly for cost reasons, to attach to a FPGA-based processing system lots of fast off-chip memory chips, the constraints imposed by the limited amount of inter-chip memory blocks in FPGA devices do require a careful design of the whole memory system.

With the relatively low granularity of memory blocks available in modern FPGAs, it is easy to organize different configurations of the memory system, with respect to the number of blocks, their depth and width. When the application data can be held entirely in the internal RAM blocks, the memory system may be organized in order to allocate sets of variables (either scalars or vectors) to several independent memories that can be accessed in parallel to feed the inputs of multi-operand datapaths at clock rate. This was exploited in [8] with a set of thirteen, 16 KByte dual-port memories, each one holding a $16 \times 16 \times 16$ 3D matrix and feeding at clock rate the inputs of a deep pipeline with 15 floating point arithmetic operators.

When external memories are needed to hold large data sets, the slow access may compromise the efficiency of the execution datapath, unless appropriate memory caching mechanisms are used to exploit the temporal and spatial locality of data. The utilization of cache memory and data prefetching for FPGA-based vector processors has been addressed in [9], where the authors study the design trade-offs for different data cache organization in a soft vector processor, while optimizing the utilization of the internal FPGA blocks of RAM. Data prefetching was exploited in order to deal with the burst access modes of modern dynamic memories, while trying to avoid filling the cache memories with surplus data.

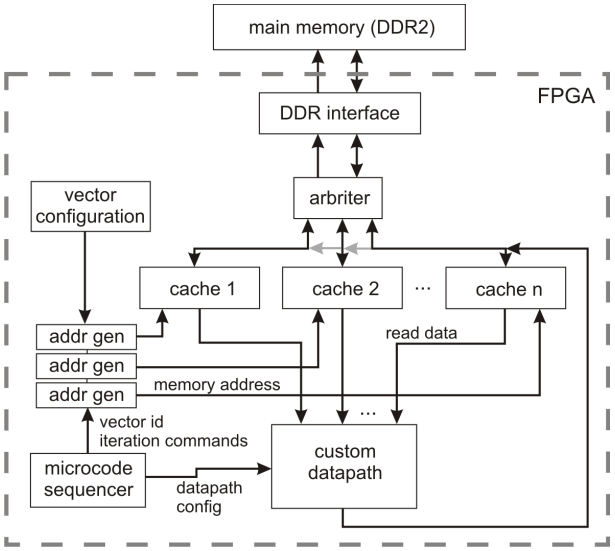

Figure 1. General organization of the cache memory system.

3. Parameterized cache memory system

In this work we extend the proposal of automatic cache generation for FPGAs [10] to build a cache memory system for vector processors, using a set of independent cache memories built with the internal SRAM block memories present in modern FPGA devices. The data width is 32 bits (for single precision floats) and each cache memory bank can be configured with different cache line size and depth. For now, only direct-mapped cache memories are supported and the whole design has been specialized for a specific family of FPGAs (Xilinx Virtex4). Besides, only 1D, 2D and 3D vectors can be handled by the address generation unit, with their elements residing in contiguous memory positions, line-by-line (for 2D and 3D vectors). This memory system is intended to implement the interface between an external dynamic memory and a custom vector processor, providing, in parallel, several data values to a custom designed pipelined datapath.

Presently this has been integrated with a simple microcode controller that issues sequences of reading commands from data vectors allocated to 4 different cache blocks. The whole system has been implemented in a Virtex4 LX80 FPGA connected to a 512 MB DDR2 memory module, integrated in a DN8000K10PSX prototyping board from the Dini Group company (www.dinigroup.com).

Figure 1 illustrates the general organization of the system and implementation details are presented in the next sections.

4. Cache memories

The configuration of each cache memory block is specified by the parameterization of a Verilog synthesizable model. Although this model do not explicitly instantiate any XILINX-specific primitives, the Verilog templates used to code the blocks of RAM memory are specific of

the Xilinx synthesis tool and may not map to similar RAM blocks present in different FPGA technologies or when using other synthesis tools.

Because the primitive SRAM blocks in Virtex4 FPGAs are 18 Kbit, the size of each cache memory must always be a multiple of 2 KByte (16 Kbit) in order to fully utilize the block memories allocated. Also, because a reading command from the DDR2 memory always returns a 32 byte block in two consecutive clock cycles (128+128 bits), the cache line size must be always defined in multiples of 32 bytes.

The associative memory was designed to be mapped into distributed memory built with lookup-tables and flip-flops, in order to reduce the read cycle (when cache hit) and the write cycle (cache miss) to a single clock period. A simple cache line replacement policy was implemented, that always substitutes the oldest written cache line. This was implemented using a FIFO for the associative memory and simple arithmetic to map each entry of the associative memory to the cache block that actually holds the data.

Two additional replacement policies can be chosen that share similar resources: LRU (least recently used) and LFU (least frequently used). A set of history registers associated with each entry of the associative memory represent either the aging of a cache line or the frequency of reading from that line, depending on the replacement policy selected.

To implement LRU, a read hit from cache line i sets its history register HR_i to the maximum value (all ones) and decrement all the registers associated to the other lines by one unit (the same happens when cache line i is replaced with new data). This is only done if HR_i has not yet the maximum value, meaning that the previous read operation was not issued from the same cache line. This avoids that repeated reads from the same cache line rapidly decrement the aging registers assigned to the other cache lines. The entry of the associative memory to be written when a replacement occurs is determined by the current values in the history registers, selecting the lowest value (meaning the oldest accessed cache line). Because the effective write into the associative memory only needs to be done when the data requested effectively arrives from the main memory, the calculation of the minimum among all the history registers can be done sequentially, within a time budget equal to the read latency of the DDR2 memory (22 clock cycles in the current implementation). Figure 2 details the logic circuit that implements the update of the history registers for LRU.

To implement the LFU technique, the selection of the cache line to be replaced is also done by choosing the cache line which history register has the minimum value. In this case, the set of history registers build a histogram representing the frequency of read accesses from each line. When a cache hit occurs and the history register HR_i of line i still does not have the maximum value, it is incremented by one; if current value is the maximum, then all the values in the history registers are divided by 2. Figure 3 presents the logic circuit that implements this mechanism.

The access to the main memory is shared by all the cache blocks instantiated in the memory system. A control module manages the read aand write requests issued from the different cache blocks and performs the reading operations, according to predefined priorities assigned to each cache block.

5. Address generator

Each cache memory block is assigned to one or more data vectors whose dimensions and locations in memory (absolute address) are known at synthesis time. Associated to each cache block, a dedicated address generator converts references to elements in a vector (the requested element indexes, for vectors up to 3 dimensions) into the absolute memory address that is then sent to the cache block. Instead of referencing absolute indexes, what would require additional arithmetic to compute the effective memory address, the references to vector elements are encoded into a small set of commands that specify an iteration over the previous reference (for example `A[i++,j]`). This translates to simple loads, additions and subtractions of constants to the address register and reduces significantly the number of control lines necessary from the microinstruction.

Table 1 presents the iteration commands implemented and the operations required to calculate the absolute memory address. Label ADDR represent the address of the

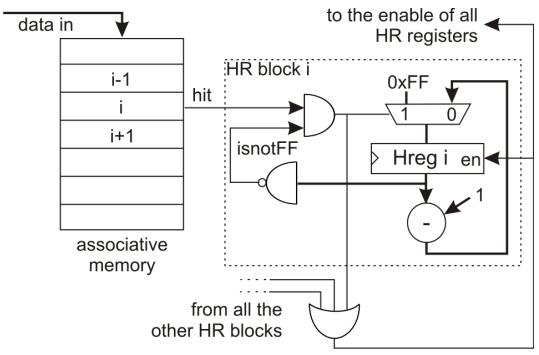

Figure 2. Logic circuit of the history registers (HR) for implementing the LRU replacement policy.

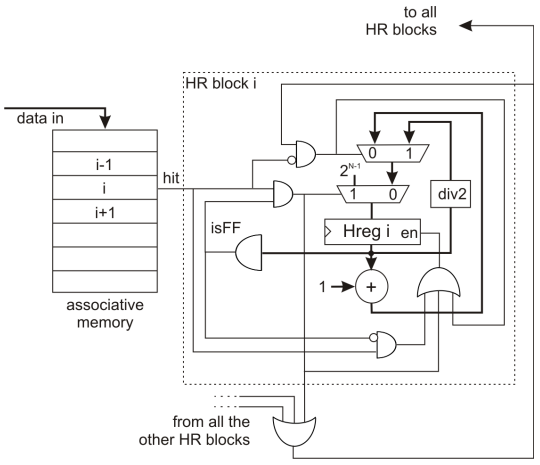

Figure 3. Logic circuit of the history registers for implementing the LFU replacement policy.

Iteration	memory address
A[i++,j,k]	ADDR+1
A[i--,j,k]	ADDR-1
A[i,j++,k]	ADDR+NI
A[i,j--,k]	ADDR-NI
A[i,j,k++]	ADDR+NI*NJ
A[i,j,k--]	ADDR-NI*NJ
A[0,0,0]	START
A[0,j,k]	START_I
A[i,0,k]	START_J
A[i,j,0]	START_K
A[NI-1,j,k]	START_I+NJ-1
A[i,NJ-1,k]	START_J+NJ*NI-NJ
A[i,j,NK-1]	START_K+NJ*NI*NK-NI*NJ

Table 1. Example of iteration commands implemented by the address generator. This considers a 3D vector A[,,] located in the main memory at address START and with NI, NJ and NK elements along each of the 3 dimensions.

last element accessed (a register) and START is a constant that represents the memory address of the first element in the vector. Three additional registers (START_I, START_J, START_K) are maintained with the address of the first element of a row along each dimension.

6. Results and conclusions

In this paper we proposed a parameterized cache memory system, aimed to increase the effective memory bandwidth for vector applications, while making use of the fast block RAMs present in modern FPGA devices. This will be later integrated into a design framework to automate the synthesis of application specific vector processors.

A first implementation was done to a Virtex4LX80-10 FPGA, including 4 independent cache blocks with LRU replacement policy. The writing process implementing the write-allocate policy has been validated in simulation but it was not yet integrated in a real hardware implementation. To issue a series of reading commands, a simple microcode sequencer sends to the cache memories a sequence of the iteration commands presented in table 1. With 4 cache memories, each one with 32 lines and 16 Kbit per line (for a total of 2 Mbit of RAM), the design uses 68% of the BRAMs, 7% of LUTs and 4% of flip-flops. This design has been sucessfully verified with a 200 MHz clock, which is the maximum frequency allowed by the interface used to access the external dynamic memories.

References

[1] Mateo Valero Roger Espasa and James E. Smith. Vector architectures: Past, present and future. In *Proceedings of the 2nd Intl. Conference on Super Computing*, pages 425–432, July 1998.

[2] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Vespa: portable, scalable, and flexible fpga-based vector processors. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 61–70, New York, NY, USA, 2008. ACM.

[3] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core cpu accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 222–232, New York, NY, USA, 2008. ACM.

[4] Junho Cho, Hoseok Chang, and Wonyong Sung. An fpga based simd processor with a vector memory unit. In *Proc. IEEE International Symposium on Circuits and Systems ISCAS 2006*, pages 4 pp.–, 2006.

[5] Christoforos Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, Computer Science Division, University of California, Berkeley, May 2002.

[6] D.A. Kozyrakis, C.E. Patterson. Scalable, vector processors for embedded systems. *Micro, IEEE*, 23(6):36–45, Dec. 2003.

[7] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proc. 30th Annual International Symposium on Computer Architecture*, pages 399–409, 2003.

[8] Filipe Oliveira, C. Silva Santos, F. A. Castro, and José C. Alves. A custom processor for a TDMA solver in a CFD application. In *ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing*, pages 63–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] J. Gregory Steffan Peter Yiannacouras and Jonathan Rose. Improving memory system performance for soft vector processors. In *WoSPS: Workshop on Soft Processor Systems*, 2008.

[10] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for fpgas. In *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, pages 324–327, 2003.