A run-time Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability

George Nikiforos, George Kalokairinos, Vassilis Papaefstathiou,
Stamatis Kavadias, Dionisis Pnevmatikatos and Manolis Katevenis
Institute of Computer Science, FORTH, Heraklion, Crete, Greece – member of HiPEAC
{nikiforg,george,papaef,kavadias,pnevmati,kateveni}@ics.forth.gr

I. Introduction

Memory hierarchies of modern multicore computing systems are based on one of the two dominant schemes: either multi-level caches with cache coherence or scratchpads with DMA functionalities. Coherent caches are customary in general purpose systems, owing to their transparency in handling data locality and communication. On the other hand, caches lack deterministic response time, and have high hardware cost, especially due to coherence complexity. Scratchpads with DMA functionalities are very popular in embedded [1] and special purpose systems with accelerators [2][3], because of their low complexity and predictable performance which is required by many real-time applications. The cost here burdens the programmers of such systems who have to deal with communication and locality explicitly.

Our work focuses on the configurability [4] of the SRAM-blocks that are in each core, near each processor, so that they operate either as cache or as scratchpad or a dynamic mix of them. We seek to provide run-time configurability to allow different programs with different memory requirements to run on the same core or even different stages of a program to adapt the underlying memory to their needs. We also strive to merge the communication subsystems required by the cache and scratchpad into one integrated *Network Interface (NI) and Cache Controller (CC)*, in order to economize on circuits.

The contributions of this work are: *(i)* we present a virtualized user-level RDMA capable NI that can serve cache and scratchpad equally well, *(ii)* we report on the complexity and performance of our scheme using an expire mental FPGA prototype. Our results demonstrate the area gains in the datapath of this integrated DMA/Cache Controller approach.

II. Architecture

The architecture and design of our NI are part of the EU-funded SARC IP project. The environment of our proposed NI is a heterogeneous [5] multicore system supporting hundreds of general purpose and special purpose cores (accelerators). The memory hierarchy of the system consists of multiple levels of conventional caches with directory-based cache coherence and scratchpad memories.

Our work presents a simple, yet efficient, solution for cache/scratchpad configuration at run-time and a common NI that serves cache and scratchpad communication requirements. The NI receives DMA commands and delivers completion notification in designated portions of the scratchpad memory. This allows the OS and runtime systems to allocate as many command buffers as desired, per protection domain, thus effectively virtualizing the NI, while providing user-level access to its functions, so as to drastically reduce latency. Relative to traditional NI's, which used their own, dedicated memory space, we improve SRAM utilization by sharing the same SRAM blocks between the processor and the NI, while preserving high-throughput operation by organizing these SRAM blocks as a wide interleaved memory. The scratchpad space can be allocated inside the L1 or L2 caches and consequently the NI is brought very close to the processor, thus reducing latency. Our NI also offers fast messages, queues, and synchronization events to efficiently support advanced interprocessor communication mechanisms; these are beyond the scope of this paper. We also assume Global Virtual Addresses and Progressive Address Translation [6].

A. Run-time configurable Scratchpad

Scratchpad space in our scheme is declared as a contiguous address range and corresponds to some cache lines that are pinned (locked) in a specific way of the cache, i.e. cache line replacement is not allowed to evict (replace) them.

Scratchpad areas can be allocated inside either L1 or the L2 caches. Most applications seem to require relatively large scratchpad sizes, so the L2 array is a more natural choice. Moreover, L2 caches offer higher degree of associativity, hence more interleaved banks. Our prototype assumes that L1 hit time is 1 clock cycle, while L2 cache hit or scratchpad access time is 4 clock cycles. The performance loss due to this increased latency is partly compensated in two ways: *(i)* our L2 and scratchpad supports pipelined accesses (read or writes) at the rate of 1 word (at random address) per clock cycle; *(ii)* configurable parts of the scratchpad space can be cacheable in the (write-through) L1 caches (our current prototype does not yet implement L1-cacheable scratchpad regions).

As in [6], owing to the use of progressive address translation, caches and scratchpad operate with virtual addresses, and a TLB need to be consulted only when going out of the node (out of L2), through the NI, to the NoC (our current prototype does not yet have a TLB in the NI). In lieu of the processor-TLB, our architecture has a small table called Address Region Table (ART) which marks contiguous address ranges either as cacheable or as scratchpad. Every scratchpad word *must*

be allocated within a cache-line whose low-order bits (cache index) are compatible with the scratchpad address. On the other hand, in the multi-way set-associative L2, the above scratchpad can be freely allocated into any of the cache ways; the ART identifies the way that is used. Identifying scratchpad regions in this way using the ART has the following advantage: a single ART entry can describe a large, contiguous scratchpad region; then all tags of this of this region, in L2, are freed (except for a single "locked" bit, used during cache accesses, when comparing all tags in a set, to tell the comparators to ignore this way); in this way the NI can use the tags for its own purposes (linked list pointers for queues of pending NI command buffers).

B. Virtualized user-level DMA

NI command buffers are DMA control areas which are allocated upon user software demand and reside in normal scratchpad regions. These buffers share the same ART entry with normal scratchpad and the distinction is made using a special bit (cache-line state), located next to the tags (set upon allocation). Any user program can have dedicated NI command buffers (DMA registers) in its scratchpad region; this allows a low-cost virtualized DMA engine where every process/thread can have its own resources. To ensure protection of the virtualized resources, we also utilize permissions bits in the ART and demand the OS/run-time system to update the ART appropriately on context switches. Moreover, the inherent support for dynamic number of DMAs at run-time, promotes scalability and allows the processes to adapt their resources on the program's communication patterns that might differ among different stages of a program.

The NI defines a DMA command format (DMA descriptor) and a protocol that should be used by the software to issue DMA operations. The DMA command descriptors include the following, common, fields: *(i)* opcode - descriptor size - DMA size, *(ii)* source address, *(iii)* destination address, The DMAs are issued as a series of store instructions, destined to a word within a marked line, that gradually fill the command descriptors, possibly out-of-order. Our NI features an automatic command completion mechanism to inform the NI controller that a new command is present. The completion mechanism monitors all the stores into marked lines and marks the written words in a bitmap located in the tag. The address tag and tag control bits in locked cache-lines are free as mentioned earlier, and thus the NI utilizes them, besides completion bitmaps, to maintain the pointers for queues of pending DMA commands that have not yet been served, either due to network congestion or high injection rate.

When serving the DMAs, the NI generates packets, along with their customized lightweight headers, that belong to one of the two primitive categories: *Write* or *Read*. The NI carefully segments the DMAs into smaller packets when they exceed the maximum packet size (256-bytes in our prototype). After the packets of the DMA command have been sent, the NI updates the DMA descriptor to signal departure (we are in the process of implementing the completion notification mechanism that will signal successful arrival of the DMA data).

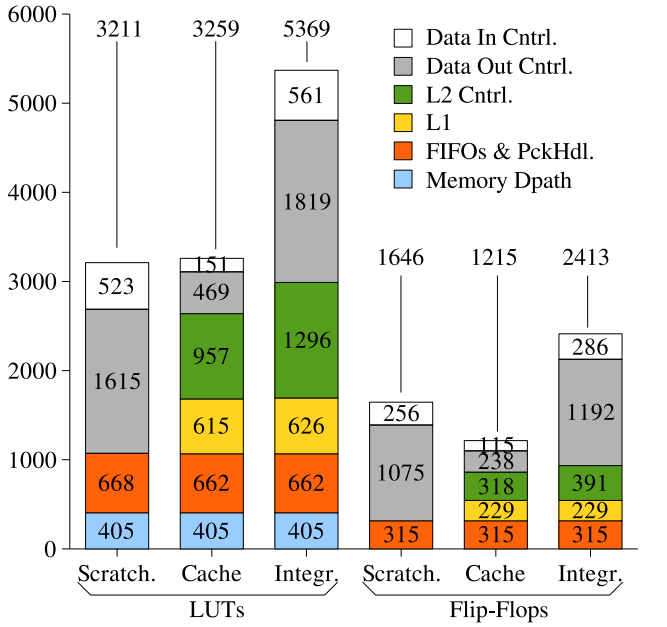

Fig. 1. LUT and flip-flop complexity of each node, excluding processor, crossbar and SRAM blocks.

Additionally, the NI offers the cache controller a dedicated set of DMA command registers in order to serve cache needs for write-backs upon replacements and fills upon misses: the same mechanisms, and the same Read and Write packets, serve DMA transfers as well as cache operations.

III. Implementation & Results

We are developing an FPGA-based prototype of our scheme in order to evaluate the performance and measure the area complexity. The design is implemented in a Xilinx Virtex-II Pro FPGA using 32-bit MicroBlaze soft-cores as processors. Our basic prototype includes two MicroBlaze processors, their 64-bit private L1 and L2 caches with our built-in NI, a 64-bit unbuffered crossbar (XBAR) for the network fabric and a DDR memory. The operating clock frequency of the system is currently 50MHz. The prototype does not yet implement any cache coherence protocol. The design and implementation details are presented in the Appendix.

Figure 1 reports the logic (LUT) and flip-flop complexity of one L2 cache/scratchpad (including L1) for one processor, together with the associated, integrated NI and cache controller; the SRAM blocks, containing tags and data, are not included in these counts. We have counted and report separately three different designs: *(i)* all SRAM operating as scratchpad only, and a NI providing DMA's; *(ii)* all SRAM operating as cache only, and a cache controller; *(iii)* our configurable cache/scratchpad with its integrated NI/cache controller. As seen, the integrated design *(iii)* has a complexity considerably lower than the sum of the complexities of the two dedicated designs, owing to several circuits being shared between the two functionalities. The circuit sharing is mostly observed on memory block datapath, the outgoing and incoming NI, and economizes more

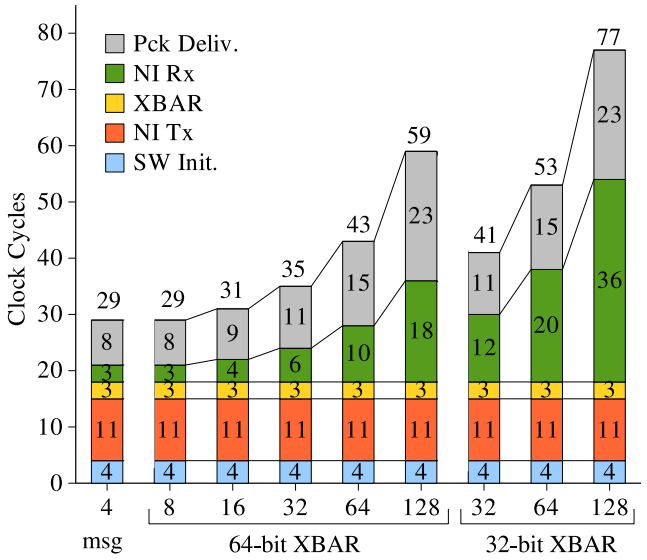

Fig. 2. DMA transfers latency breakdown, as a function of data size (bytes)

than 15% in hardware complexity. We expect that the shared circuits will increase as we incorporate more advanced cache mechanisms such as multiple outstanding cache misses and coherence.

Figure 2 presents the latency breakdown of several RDMA-write transfers. The SW initiation cost, the NI transmit latency and the XBAR latency of every DMA are constant under zero network-load conditions – the outgoing path implements cut-through. The latency for the reception of the packets and delivery of the payload in memory are commensurate to the size of the transfer – the incoming path implements store-and-forward, in order to check packets' CRC for errors. Minimum-sized messages and DMAs of 4 and 8-bytes have the same end-to-end latency of 29 clock cycles, this is attributed to the memory bandwidth which is 8-bytes/cycle. Larger DMAs cost significantly more cycles, e.g. a 128-byte DMA costs 59 cycles, on our 64-bit XBAR while assuming a 32-bit XBAR, the cost is 77 cycles. The NI transmit path has a latency of 11 cycles where 4 of them are attributed to the outgoing FIFO that serves both as a NoC buffer but also as clock domain synchronizer – crosses cache and network clock domains.

IV. Conclusions and Future Work

The development of our prototype and the initial evaluation of our integrated *Network Interface and Cache Controller* proves the feasibility of this approach and the existence of circuitry that is shared between the network interface and cache controller. DMA latency is at considerably low levels. We are now working towards adding extra features in the NI and merging them with more advanced cache functionalities like multiple outstanding cache accesses and coherence.

V. Acknowledgements

This work is supported by the European Commission in the context of the SARC integrated project #27648 (FP6). We also thank, for their assistance in designing the architecture and in implementing the prototype: Dimitris Nikolopoulos, Alex Ramirez, Georgi Gaydadjiev, Spyros Lyberis, Christos Sotiriou, Euriclis Kounalakis, Dimitris Tsaliagos, Xiaojun Yang, and Michael Ligerakis.

References

[1] Rajeshwari Banakar, Stefan Steinke, Bo sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *In 10th International Symposium on Hardware/Software Codesign (CODES), Estes Park*, pages 73–78. ACM, 2002.

[2] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Develeopment*, 49(4/5):589–604, 2005.

[3] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, 2003.

[4] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. *Proceedings of the 27th International Symposium on Computer Architecture, 2000.*, pages 161–171, 2000.

[5] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, Nov. 2005.

[6] M. Katevenis. Interprocessor communication seen as load-store instruction generalization. In *The Future of Computing, essays in memory of Stamatis Vassiliadis, K. Bertels e.a. (Eds.), Delft, The Netherlands*, pages 55–68, September 2007.

[7] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *HPCA'96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

[8] Kenneth C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

VI. Appendix

This appendix provides details about the implementation of the architecture described in section II. Our processor is the 32-bit MicroBlaze soft-core with a basic 5-stage pipeline. The memory hierarchy consists of L1 and L2 private caches. Usual L1 cache sizes range around 16 to 64 KBytes, and their associativity is 2 to 4 ways, with 64-byte cache-lines. Our implementation has scaled down the L1 cache size to 16KB, direct-mapped with 32-byte cache-lines. L1 cache is write through with no-allocate on store miss policy. L2 caches, on the other hand, are usually much larger, with sizes beyond 1MB, their associativity is up to 16-ways, and the cache-line size up to 128 bytes. Scaling down again, we have designed a 64KB, 4-way set-associative write-back L2 cache with 32-byte cache-lines. Each data way is 64-bits wide. L2 cache controller supports a single outstanding miss, multiple hits under single miss, and has a single entry deferred write buffer which supports bypassing. The L2 controller serves cache misses using DMA primitives: a cache fill request is treated as a DMA read, and a cache write back as a DMA write. This is one of the places where the cache and DMA controllers get integrated.

The default set-associative cache organization is quite power-hungry, since it requires all the cache ways (tags + data) to be probed in parallel. To decrease consumption we adopted sequential tag accesses to determine hits. To reduce hit time, we used Way Prediction [7][8]; this, also, further reduces energy consumption, and it allows tag arrays to be placed together, "one under the other", in a single dual ported memory

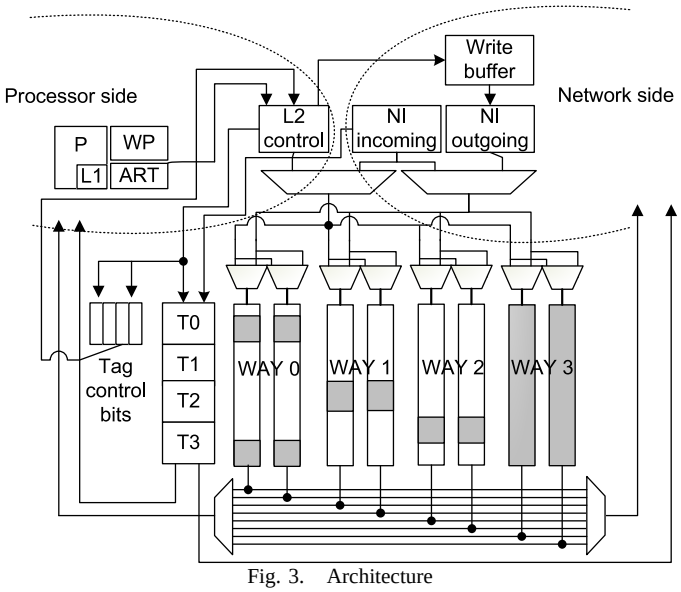

Fig. 3. Architecture

block as presented in Figure 3. While most schemes try to use and benefit from PC values to predict a way, we implemented a sequential access approach which uses signatures of the address tags. Each signature is created by bitwise XORing the 3 MS bytes of address tag. Instead of probing all tags in parallel and comparing against the MS bits of the address we use a memory block to store the signatures from every address tag. The comparison of the signatures with the actual address, which is done in parallel with L1 tag matching, gives an indication for the ways that are highly possible to produce a hit. These ways are then sequentially probed to verify that there is no false positive hit. The signature format is carefully selected so that no false negatives occur. Way Prediction generates a 4 bit mask (one bit for each way) in parallel with L1 tag matching (Figure 3), so when L1 decides if the processor request must be forwarded to L2 cache, the way prediction mask helps in decreasing L2 tag access time.

When an L2 cache miss occurs, a cache line must be selected for replacement. In normal caches, the replacement policy has to select among a fixed number of cache-lines per set (associativity number). In our configurable cache/scratchpad the replacement policy uses random decision among non (locked) scratchpad areas (grey areas in Figure 3), since scratchpad lines are not evictable. The valid, locked, and dirty bits are maintained in separate SRAM blocks, relative to the tag block. For the former, all ways are accessed in parallel, gaining access time when replacement decision must be made. The array that keeps the dirty bits has two ports in order to support back to back stores and loads – stores need to set the "dirty" bit.

L2 local memory consists of multiple interleaved banks as shown in Figure 3, for throughput purposes. The FPGA prototype emulates a real system with 8 single ported banks, using 4 dual ported SRAMs, artificially refusing to serve to simultaneous accesses to a would-be single bank. The ports are shared between incoming and outgoing NI on one hand and the processor on the other. These agents compete for accesses to the L2 banks, causing conflicts which are handled by an arbiter. The peak throughput is 128bits/cycle. The implementation supports pipelined back to back stores and delayed or deferred stores when immediately followed by loads.

When a packet reaches its destination, the incoming network interface has to detect if the packet refers to cacheable or scratchpad area. In case it refers to cacheable area the memory access (write) has to follow wrap-around and critical word first policy for size of one cache line, in contrast, for scratchpad areas, the data are stored in contiguous addresses starting from destination address and use the size referred in the incoming packet.

The communication between the different cores and the main memory is accomplished through an unbuffered crossbar with a simple round robin scheduler.