Hardware Primitives for the Synthesis of Multithreaded Elastic Systems

G. Dimitrakopoulos, I. Seitanidis, A. Psarras, K. Tsiouris
Electrical and Computer Engineering
Democritus University of Thrace, Xanthi, Greece

P. M. Mattheakis
Mentor Graphics
Fremont, USA

J. Cortadella
Universitat Politècnica Catalunya
Barcelona, Spain

***Abstract*—Elastic systems operate in a dataflow-like mode using a distributed scalable control and tolerating variable-latency computations. At the same time, multithreading increases the utilization of processing units and hides the latency of each operation by time-multiplexing operations of different threads in the datapath. This paper proposes a model to unify multithreading and elasticity. A new multithreaded elastic control protocol is introduced supported by low-cost elastic buffers that minimize the storage requirements without sacrificing performance. To enable the synthesis of multithreaded elastic architectures, new hardware primitives are proposed and utilized in two circuit examples to prove the applicability of the proposed approach.**

I. Introduction

Elastic systems offer new degrees of freedom to the designer by relaxing the strict global event scheduling requirements of conventional synchronous designs and enabling the dynamic scheduling of operations based on the availability of the corresponding data [1]. This flexibility has been exploited in SoC IP integration using elastic IP wrappers [2], in hardware units synthesized from dataflow programming models [3], in massively parallel processors [4], [5], as well as in elastic coarse-grained reconfigurable arrays to schedule operations dynamically via elastic control [6].

Synchronous elastic circuits are behaviorally equivalent to conventional synchronous circuits with respect to the trace of valid data observed at the inputs and outputs of the computation units and communication channels, as shown in Figs. 1(a) and 1(b). However, the cycles at which the valid data appear may vary significantly. Multithreaded elastic systems allow independent threads to re-use the datapath of the baseline elastic circuit thus maximizing hardware utilization and minimizing the idle cycles that naturally arise from variable latency operations and data-dependent branching. Fig. 1(c) shows an example of the multithreaded elastic version of the elastic flow presented in Fig. 1(b) where the empty slots are filled with valid data that belong to a second independent thread.

Threads operate in a time-multiplexed manner and the selection of the thread that proceeds to execution involves arbitration among the active threads. Threads may share the datapath either in a fine-grained manner by changing the active thread on cycle-by-cycle basis or in a coarse-grained manner that allows each thread to complete a larger set of computations before moving to the next one [7].

In this paper we minimize the number of buffers required to support multiple threads in an elastic system by sharing buffers across threads. The proposed multithreaded elastic buffers can be designed in a modular manner either with regular edge-triggered flip flops or level sensitive latches. New multithreading and control operators, such as join, fork, branch and merge are introduced that can be employed in the synthesis of generic elastic architectures. In the same context, we define and implement a new thread synchronization primitive (barrier) that can be used for the high-level synthesis of multithreaded algorithms. Two examples, an MD5 cryptographic hash function and a multithreaded pipelined processor, prove the applicability of the proposed methodology.

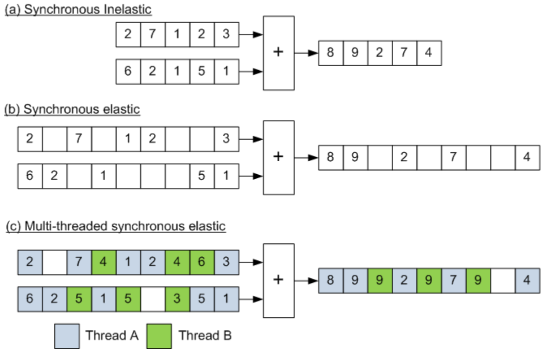
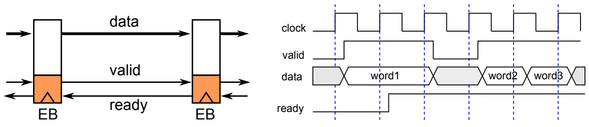


Fig. 1. Single and multithreaded elasticity versus inelastic operation.

II. Baseline Elastic Protocol and Circuit

A baseline elastic channel carries data signals and two handshake signals (valid and ready) that implement the elastic protocol, as shown in Fig. 2(a). The elastic buffers (EBs) implement the handshake protocol by replacing any simple data connection with an elastic channel. When an EB can accept an input, it asserts its ready signal upstream; when it has output available, it asserts valid downstream. When valid and ready are both asserted, a data transfer occurs, as shown in Fig. 2(b). In its simplest form, the forward and backward latency of the handshake signals to the adjacent stages is one cycle. With these conditions, any EB requires a minimum storage capacity of two data items [8] and it can be in three possible states: EMPTY, HALF and FULL depending on the items stored in the buffer.



(a) (b)

Fig. 2. Single-thread elastic protocol and buffers.

The orchestration of the computation in elastic architectures requires additional primitive control operators such as join and

978-3-9815370-2-4/DATE14/©2014 EDAA

fork modules that handle data convergence or split and program control flow operators such as "*if-then-else*" statements using the branch and the merge operators, shown in Fig. 3.

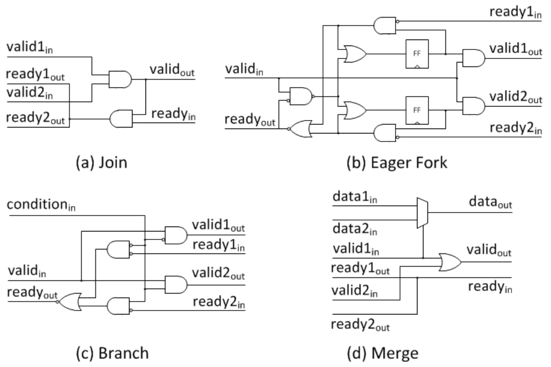


Fig. 3. The operators used for the synthesis of elastic architectures.

III. Multithreaded Elastic Protocol and Buffers

Multithreaded elastic control assumes that the elastic modules and channels operate in a time-multiplexed manner allocating time slots to threads. A multithreaded elastic channel carries the data of only one thread at each cycle and as many pairs of handshake wires (valid(i)/ready(i)) as the number of concurrent threads supported by the system. Among the active threads only one uses the channel, i.e., only one valid(i) signal is asserted per cycle. An arbiter is responsible for selecting the active thread after taking into account which threads are ready downstream.

A baseline multithreaded elastic buffer (MEB) can be built by replicating one EB per thread and adding an arbiter and a multiplexer, as shown in Figure 4 for the case of 3 threads. The use of a 2-slot EB per thread, called full MEB, is an expensive solution since the available resources (EBs in this case) are effectively replicated per thread.

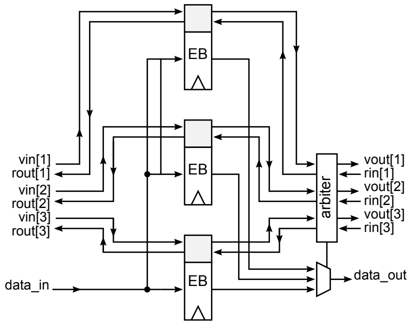
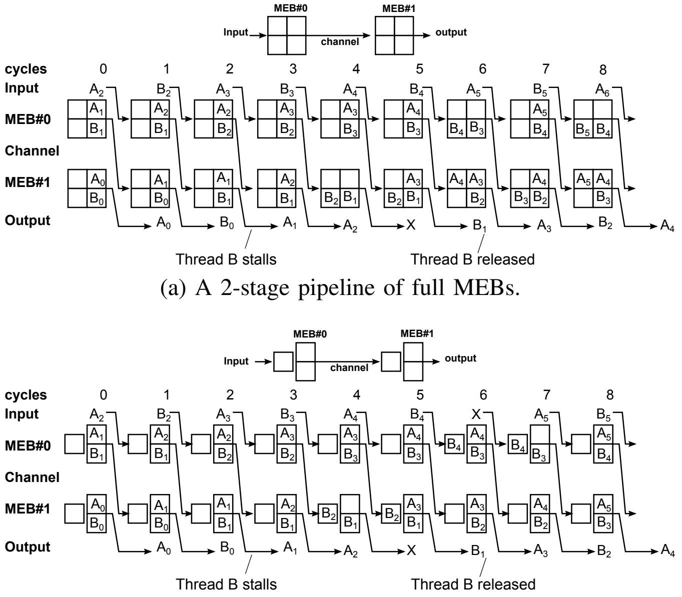


Fig. 4. The micro-architecture of a 3-thread MEB using multiple EBs.

A. Reducing the capacity of MEBs

Let us assume that the system can support S threads and that M threads are active in an elastic channel in a particular cycle. If $M = 1$ (only one thread is active), a 100% throughput can be achieved for the active thread. If the active thread stalls, the two available slots will be occupied, while the slots of the remaining $S - 1$ threads will be empty.

When M threads are active, with $2 \leq M \leq S$, each thread will receive a throughput of $1/M$. In this case of uniform utilization, each thread will use only one buffer out of the two available per thread since it will be accessed (read or written) once every M cycles. Only when a thread stalls, it will use its second auxiliary buffer. If every thread can use its secondary buffer independently of the rest it can always enjoy full throughput. If we relax this requirement we can build a reduced MEB using only $S + 1$ slots. Each thread owns a single slot (S in total) which is enough in the case of uniform utilization when each thread delivers a throughput of $1/M$. Also, when a single thread uses the channel without any other thread being active or blocked, i.e., $M = 1$, it receives full throughput and in the case of a stall it may use the extra slot available in the reduced MEB. *The extra slot is shared dynamically by all threads, although only one of them can occupy it at a time.*



(a) A 2-stage pipeline of full MEBs.

(b) A 2-stage pipeline of reduced MEBs.

Fig. 5. Elastic flow on MEB pipelines.

Figure 5 depicts an example of control flow for an elastic channel that supports two threads using both full and reduced MEBs. In the first cycles, each thread receives $1/2$ of the throughput per channel and utilizes only one buffer slot. In those cycles, the shared auxiliary registers are not utilized. The shared slots are used for accommodating the stalled data items of thread B. If the backpressure for B reached the input of the pipeline, as done in Fig. 5(b), injection for thread B stops and only data for thread A enter the system. This situation is the only one in which the difference between the full and the reduced MEB arises: when all threads, except one, are blocked and the shared buffers of all pipeline stages up to the source are utilized by a blocked thread, then the only active thread will obtain 50% of throughput, since it effectively sees only one available slot per channel. Full MEB, on the other hand, will allow the active thread to fully utilize the channel. The occurrence frequency of this effect depends on how often all but one of the threads are stalled, which is application dependent, and on the number of cycles it takes the stall to propagate to the source of the pipeline.

IV. Multithreaded Elastic Primitives

In this section, we describe in detail the implementation of (a) the proposed reduced MEB, (b) the multi-threaded variants of the data redistribution and control primitives as well as (c) the thread synchronization primitive that are employed in the synthesis of multithreaded elastic circuits.

A. Reduced Multithreaded Elastic Buffer

The reduced MEB can be designed using the datapath shown in Figure 6 that consists of a single register per thread along with an auxiliary register that is dynamically shared by all threads.

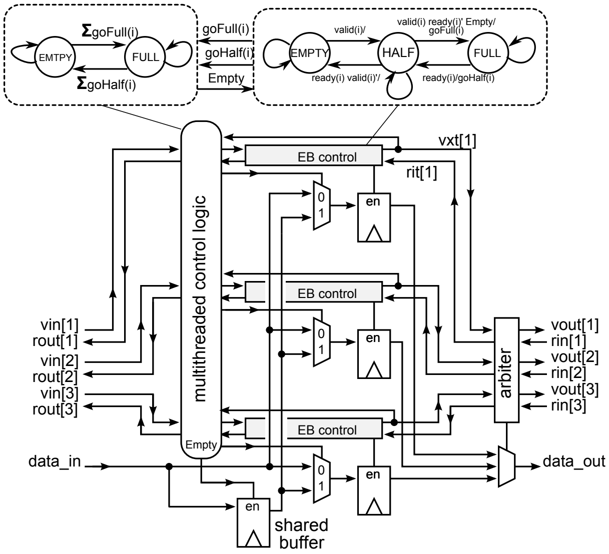
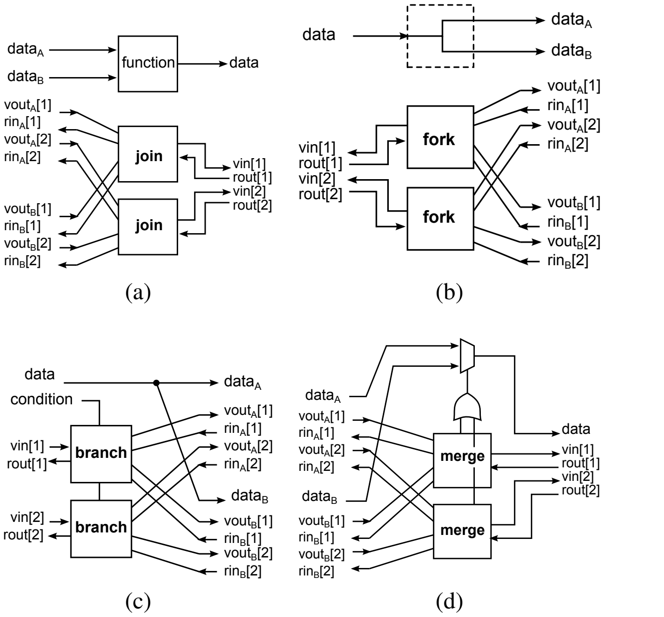


Fig. 6. The reduced 3-thread MEB with a dynamically shared buffer.

The elastic thread control unit copies S times the control logic of a single EB that implements the 3-state FSM shown in Fig. 6, allowing each thread to be in EMPTY, HALF or FULL states. The elastic thread control tracks the state of each EB by inspecting the additional goFull and goHalf signals and guarantees via the Empty output signal that only one of them will move to the FULL state. This is needed since in the reduced MEB only one thread is allowed to store two data items in case of a downstream stall by using the shared buffer. A two-state FSM associated with the shared buffer tracks this condition and produces the Empty signal that allows the transition of only one EB control from HALF to FULL state.

When a new data item arrives that belongs to the i-th thread which is in EMPTY state, it is stored at the main register of the i-th thread and moves to the HALF state. The threads in the HALF state are ready to accept new data, as long as no thread is in the FULL state. If this is the case and new data arrives, three operations take place in the same cycle: (a) the new data item is stored to the shared buffer, (b) the corresponding thread moves to the FULL state, and (c) all threads that were in the HALF state stop being ready to accept new data.

When the arbiter selects a thread that is in HALF state, its data is dequeued from the thread's main register and the thread returns to the EMPTY state. On the contrary, if the selected thread was the only one in the FULL state having stored two words in MEB (at the main register and the shared buffer), then it should move to the HALF state after reading the data from the main register. During this state transition, the main register of the thread should be refilled by the data stored in the shared buffer. The shared buffer cannot receive a new word in the same cycle since its availability will appear on the upstream channel in the next clock cycle.



(a) (b)

(c) (d)

Fig. 7. Multithreaded versions of the elastic control operations (a) M-Join and (b) M-Fork. (c) M-Branch and (d) M-Merge.

B. Multithreaded Elastic Control Operators

The multithreaded versions of the control primitive operators join and fork named M-Join and M-Fork are designed by using multiple join and fork modules, equal to the number of supported threads, as shown in Figure 7. The handshake signals of both inputs are first gathered per thread and then connected to the baseline single-thread join and fork operators.

In the case of M-Branch operator the movement of data and its associated elastic control is dictated by a condition flag, as in the single-thread case. The active valid bit of the input elastic channel reveals to which thread the condition corresponds to, as shown in Figure 7(c) for the case of two threads. The M-Merge unit merges the data streams created by M-Branch to a single multithreaded elastic channel. Figure 7(d) shows the implementation of M-Merge for two threads. Which path and which thread will be active are dictated by the operation of M-Branch that distributes the control information per thread and per path. Out of the two paths, only one will be active. Therefore, two baseline merge units suffice to merge the control information per thread.

C. Thread Synchronization

Thread barrier synchronization acts like a coordination mechanism that forces the threads that participate in a multithreaded elastic system to wait until each one of them has reached a certain phase of the algorithm's execution. Once all threads have reached the barrier carrying valid data, they are all permitted to continue past the barrier.

The implementation of a barrier [9] that is compatible to the multithreaded elastic protocol is shown in Figure 8. Initially

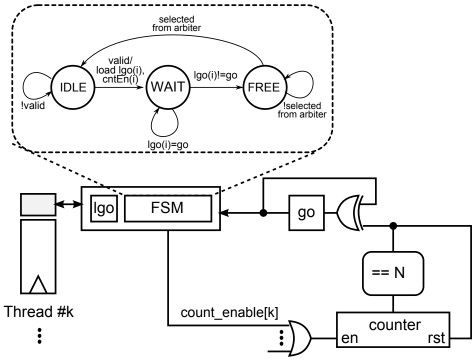


Fig. 8. Multithreaded elastic thread synchronization primitive (barrier).

all threads that do not have valid data stay in IDLE state. Once a new data item arrives for a certain thread, this thread moves to the WAIT state and increments the counter. If this was the last thread to arrive, the counter will reach the barrier's limit and it will be reset to zero. At the same time the value of the global go flag will be flipped thus signaling the arrival of all the threads in the barrier and their transition to the FREE state. On the contrary, if the arriving thread was not the last one, thread's state remains unchanged. All threads remain at the FREE state until they are selected by the arbiter. Once selected, they move to IDLE state waiting for the barrier to re-open.

V. Design Examples

The proposed design methodology have been verified in two different cases of multithreaded elastic circuits: (a) an MD5 hash function and (b) a multithreaded pipelined processor.

A. MD5 Cryptographic Hash Function

The MD5 algorithm takes as input a plaintext of arbitrary length and, after processing it in fixed-length blocks, it produces a 128-bit hash value. The algorithm comprises of 4 rounds of 16 steps each. The 16 steps of each round are fully unrolled and implemented in a single cycle, although they could have been pipelined with minimum changes due to elasticity. Each thread requires four rounds to complete the execution of MD5. Since MD5 requires a different configuration for each round, all threads need to synchronize before moving to the next round. This is achieved by using a barrier, which blocks the data flow after the output buffer. When all threads have been processed and reached the barrier, the data flow is released, allowing the round counter to be incremented.

B. Pipelined Multithreaded Elastic Processor

In the second example, we designed a multithreaded elastic processor that implements the instruction set of [10] and allows each thread to execute its code independently by sharing the functional units of the datapath. Every pipeline register has been replaced by a MEB that selects independently at each stage which thread to promote for execution. Each thread sees a different copy of the register file and has a private program counter. All threads are eligible to move forward in the pipeline as long as they contain a valid instruction. The instruction and data memory as well as the execution units are considered variable latency units.

C. FPGA implementations

Table I summarizes the implementation results (area/delay) of the MD5 hash function and the multithreaded processor when built with full and reduced MEBs in the place of pipeline registers. For the processor, the number of block RAMs used for data and instruction memories as well as the multithreaded register file and the DSP blocks are not included and they are the same in both cases by construction. The reduced MEBs save in average 15% in both examples without sacrificing either clock frequency or performance in terms of throughput. Actually, the slightly higher clock frequencies achieved are a result of the smaller wiring delays due to lower area. The savings in the processor are larger than in MD5, since it has a larger ratio of MEB area vs combinational logic area. If we increase the number of threads to 16 the average savings rise above 22%.

TABLE I
FPGA implementation results of the 8-thread design examples.

Design	Full MEB		Reduced MEB	
	Area (LEs)	Freq. (MHz)	Area (LEs)	Freq. (MHz)
MD5 hash	12780	11	11200	12
Processor	6850	60	5590	68

VI. Conclusions

The unification of elasticity and multithreading opens up new possibilities to the designer for deriving systems that can support variable latencies and multiple threads under a fully distributed and self-contained data flow control. At the same time, the cost of buffering that may dominate such systems is significantly reduced by allowing sharing of buffers across threads, while still offering deadlock free operation in an efficient and modular manner. Newly derived multithreaded elastic control primitives as well as the thread synchronization barrier enable the automated synthesis of complex algorithms to their multithreaded elastic equivalent circuits.

References

[1] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *IEEE Transactions on Computer-Aided Design*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.

[2] L. Carloni and A. Sangiovanni-Vincentelli, "Coping with latency in soc design," *IEEE Micro, Special Issue on Systems on Chip*, vol. 22, no. 5, pp. 24–35, 2002.

[3] D. Capalija and T. Abdelrahman, "Towards Synthesis-Free JIT Compilation to Commodity FPGAs," in *Int. Symp. on Field-Programmable Custom Computing Machines*, 2011, pp. 202–205.

[4] M. Butts, A. M. Jones, and P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2007, pp. 55–64.

[5] R. Panda and S. Hauck, "Dynamic Communication in a Coarse Grained Reconfigurable Array," in *19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 25–28.

[6] Y. Huang and et al., "Elastic CGRAs," in *Proc. of the ACM/SIGDA Intern. Symp. on Field programmable gate arrays*, 2013, pp. 171–180.

[7] T. Ungerer, B. Robic, and J. Silic, "A survey of processors with explicit multithreading," *ACM Computing Surveys*, vol. 35, pp. 29–63, 2003.

[8] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency insensitive design," *IEEE Transactions on Computer-Aided Design*, vol. 20, no. 9, pp. 1059–1076, 2001.

[9] G. Andrews, *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison-Wesley, 1999.

[10] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP Block Based FPGA Soft Processor," in *Proc. of the Intern. Conf. on Field Programmable Technology, Seoul, Korea*, Dec. 2012, pp. 151–158.