Memory centric thread synchronization on platform FPGAs

Chidamber Kulkarni
Xilinx Inc
San Jose, Ca
Chidamber.Kulkarni@xilinx.com

Gordon Brebner
Xilinx Inc
San Jose, Ca
Gordon.Brebner@xilinx.com

Abstract

Concurrent programs are difficult to write, reason about, re-use, and maintain. In particular, for system-level descriptions that use a shared memory abstraction for thread or process synchronization, the current practice involves manual scheduling of processes, introduction of guard conditions, and clocking tricks, to enforce memory dependencies. This process is tedious, time consuming, and error-prone. At the same time, the need for a concurrent programming model is becoming ever essential to bridge the productivity gap that is widening with every manufacturing process generation. In this paper, we present two novel techniques to automatically enforce memory dependencies in platform FPGAs using on-chip memories, starting from a system-level description. Both the techniques utilize static analysis to generate circuits for enforcing these dependencies. This paper will investigate these two techniques for their generality, overhead in implementation, and usefulness or otherwise for different application requirements.

1. Introduction

Concurrent programs allow multiple computations to occur simultaneously in cooperation with each other. How this cooperation between concurrent computations is expressed and eventually realized in platform FPGA implementations determines whether it is advantageous to exploit available concurrency for a given application or not. In this paper, we focus on shared memory abstraction for such cooperation between concurrent computations. In particular, we investigate mapping of synchronization mechanisms for a shared memory abstraction on to platform FPGAs for networking applications.

The search for application-specific solutions with ever decreasing time-to-market is pushing system designers away from the risky time-consuming ASIC design process towards programmable platforms, such as FPGAs. An example of platform FPGA is the Xilinx Virtex-II Pro family of programmable logic devices that include hard IP cores such as the PowerPC microprocessor and RocketIO serial transceivers, and true dual-ported on-chip embedded block RAM (BRAM) memories [4]. The current state-of-the-art design flow for platform FPGAs in the networking domain involves starting with HDLs, which are not well suited for describing systems. Thus a productivity gap is widening with every new process generation. To close this gap, we need methods and tools that can transform higher-level concurrent semantics into HDL implementations (at the register-transfer level). A primary goal of this work is to enable a higher abstraction for mapping applications in the networking domain to platform FPGAs. Current shared memory abstractions based on locks and mutual exclusions are difficult to use, scale, and generally result in a tedious and error-prone design process.

To alleviate some of these difficulties, *transactional memories* have been proposed [1]. Transactional memories allow programmers to define customized read-modify-write operations to apply to multiple independent memory locations [1]. The well known advantages of using a transactional memory are: a lock-free programming abstraction, avoiding problems from deadlock, priority inversion, and convoying, and the ability to roll back after exceptions and timeouts. Software transactional memories [2,3] address the above-stated problems using language constructs and operating system support, whereas hardware transactional memories use additional logic to carry out the guarded (atomic) operations. The main drawback of software transactional memory is the overhead incurred by operating system (or virtual machine) run time. For hardware transactional memories [1], the area overhead comes mainly from the additional memory needed to store the transactions, the memory state, and the logic to compare and copy or discard. To date, all such proposals have been implemented in simulations. Hence no real area overhead data is available for hardware transactional memory implementation. In addition, existing work in transactional memory has not leveraged programmable fabrics such as FPGAs for customizing memory-associated control for a given application since their focus has been on pre-defined instruction-set processors.

In this context, this paper investigates two techniques to implement memory dependence enforcement in memory controllers associated with on-chip BRAM memories in FPGAs. The main difference between our approach and that of related existing work [1,2,3] is that: our approach handles the inter-process communication/synchronization problem as a memory dependence enforcement problem; and our approach does not store copy(ies) of memory location(s) currently being modified, so we do not perform roll-back operations. The latter condition results in exclusion of a class of applications that require strict atomicity to be enforced in the event of external events (or interrupts). However, for our application domain we believe that this is a reasonable restriction. From a programming model view point, the programmer does not need to use locks (and mutual exclusions) and the blocking semantics are enforced by the implementation, thus retaining the advantages of a transactional memory. In addition, deadlocks are identified statically since the user explicitly specifies producer(s) and consumer(s).

The remainder of the paper is organized as follows: Section 2 introduces the experimental language, hic, in which we implement our new techniques. Section 3 discusses the semantics and tool flow aspects of starting from a system-level description in hic and generating the desired implementations onto platform FPGAs. Section 4 presents the results and related discussion. Section 5 presents a brief overview of related work. We end the paper with the main conclusions and directions for future work.

2. Background

Hic is a concurrent asynchronous language for describing networking applications. At the core of this language are two main concepts: concurrency expressed in threads, and a logical global shared memory model that represents a tub of packets (or cells). The global shared memory is presented using a pre-defined data type called *message.* Threads receiving and transmitting at the network interface keep writing and reading messages, one at a time. Threads performing computation have at most one *message* in-flight and each thread runs to completion processing a *message*. This logical global shared memory is then mapped on to a physically distributed on- and off-chip memory organization as is found on FPGAs. This mapping is done based on a memory access graph and an operation order graph that is generated by the front-end compiler. This mapping is not the focus of the current paper and hence we will not discuss it further. Each thread has an internal structure similar to a high-level language. It comprises of: declaration of internal variables, currently the three supported types are: integer, character, and a user defined type (eg: with fixed bit width or a union of existing types); statements performing operations on these variables, support for conditions, state machines (case statements), and looping constructs (for and while loops) are present. In addition, there are four pragmas provided for describing additional aspects of system that will help compiler perform useful (and necessary) optimizations. The four pragmas describe interfaces (eg: Gigabit Ethernet), constants (eg: host address), producer data, and consumer data. The latter two are specified by the user to indicate inter-thread memory dependencies that need to be obeyed. This in turn helps the compiler to explore ways to implement this dependency and the user can select different implementations based on constraints s/he sets for the specified system. This is the focus of current paper, namely exploring how to implement inter-thread memory dependencies. It is important to note that in our parlance thread means a hardware thread, that is, each thread is synthesized in to logic. This is based on the concept of multi-threading in logic as introduced by Brebner [5].

Figure 1 shows a pseudo-example that uses the producer and consumer pragmas that are used to specify the inter-thread memory dependencies. Here *x1* is the producer and *y1* and *y2* are the consumers. In thread t1, the pragma #*consumer*{mt1, [t2,y1], [t3,z1]} prior to the assignment says that x1 is consumed in thread t2 by y1 and in thread t3 by z1. Similarly, in thread t2 and t3, the #*producer*{mt1, [t1,x1]} indicates the producer of particular data. The additional identifier, *mt1*, in the pragmas is used to identify multiple dependencies on same variable in threads. Based on this language background, we will now introduce two techniques that we intend to use as an implementation for given inter-thread communication and the resulting memory dependencies. It is necessary to recognize that the particular syntax used here is not central to our techniques but is used to make analysis easier for the front-end tools. In practice, one can use standard compiler use-def analysis [7] and other lifetime analysis methods [9] to extract producers and consumers from a given specification. The next section describes the tool flow starting from hic, and the implementation of aforementioned techniques.

```
thread t1 ()
{
    int x1, xtmp, x2;
    ...
    #consumer{mt1,[t2,y1],[t3,z1]}
    x1 = f(xtmp, x2);
    ...
}
```

```
thread t2 ()                      thread t3 ()
{                                 {
    int y1, y2;                       int z1, z2;
    ...                               ...
    #producer{mt1,[t1,x1]}            #producer{mt1,[t1,x1]}
    y1 = g(x1, y2);                   z1 = h(x1, z2);
    ...                               ...
}                                 }
```

Figure 1 Example hic description

3. Memory centric thread synchronization

The design flow used in this paper comprises describing an application in hic, from which a RTL HDL description is generated. This RTL code is then is fed into standard synthesis, place, and route tools to obtain the final implementation of the design on a FPGA device.

We describe two techniques to implement inter-thread synchronization based on thread-level analysis (here, based on hic). In the hic front-end compilation, a series of synthesis steps are applied that transform the hic threads into state machines. These steps are well researched in the behavioral synthesis community [6]. These state machines are cycle accurate and we have knowledge of the particular state where memory accesses happen. However the underlying assumption until this point is that the memory accesses are all single cycle and happen in that particular state (in the FSM). This assumption has been critical for applying different transformations to either deduce a memory organization, share operations, and to build cycle accurate FSMs. However in practice to enforce memory dependency, the user has to either syntactically provide enough hints or architect so to explicitly schedule these concurrent operations. This process is tedious and error-prone.

From the hic description, we derive the producer-consumer relationships as specified by the user. Thus with each thread there is an associated list that indicates whether there is a producer and/or a consumer in the particular thread (in a particular state). This list is a subset of the total memory needed by the particular thread and it aids in memory allocation and assignment decisions. In this design process the user makes memory allocation decisions based on the memory size analysis and a partial order of operations. We term the order as a partial order since at this stage we do not know which memory accesses are going to take multiple cycles due to dependencies on other threads. However the memory allocation process takes into account available physical memory size (eg: BRAM size of 18 Kb) and number of ports (eg: dual ports on each BRAM). Different approaches to similar problems have been researched in the past [8,9,10]. Given such a memory allocation, we have enough information in the above-mentioned lists about which memory locations have dependencies that need to be enforced.

Based on a list of these relationships we can now generate points in the design where we need to insert memory dependence enforcement on a per-BRAM basis. We will now describe two techniques that are used to implement memory dependence enforcement (in memory controller) based on the above analysis information.

3.1 Arbitrated memory organization

The first technique used to enforce memory dependencies is termed as arbitrated memory organization. An instance of the base architecture of such a memory organization is shown in Figure 2. In the following, we now describe how we map inter-thread memory dependencies on to such an architecture starting from hic.

Based on the producer-consumer relationships and memory allocation (as determined by the user) obtained from static analysis we can automatically generate the allocation and access to particular ports of a BRAM (on-chip block RAM memory) that is common to two or more threads. The port details of how this gets implemented are given next.

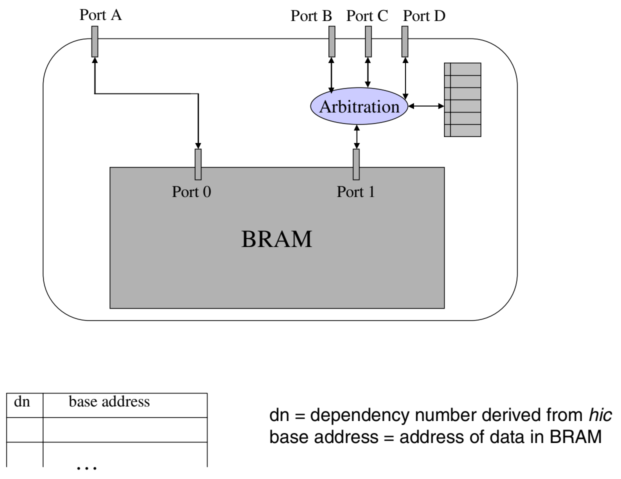
Port A
Port B
Port C
Port D
Arbitration
Port 0
Port 1
BRAM
dn
base address
…
dn = dependency number derived from *hic*
base address = address of data in BRAM

Figure 2 Arbitrated memory organization

The actual implementation of the arbitration scheme comprises the use of a wrapper around the physical BRAM structures as shown in Figure 2. This wrapper comprises two additional ports apart from the standard dual ports of a BRAM. Of these two additional ports, one is a read port (labeled C) and the other a write port (labeled D). The other two ports, labeled A and B, perform both read and write as in a normal BRAM. Of the two standard ports one port has a direct access to the physical BRAM port. The remaining three have priority-based access, wherein the write port (port D) gets highest priority, the read port (port C) gets second priority, and the remaining standard port has lowest priority. In addition, the access to port C and port D is arbitrated. This is because there can be more than one thread as a client on these ports, as in the case of multiple producer and consumers sharing the same memory (and ports). Thus they share separate buses. This arbitration is not shown in Figure 2. In such a case, there will be an additional layer of multiplexer(s) on port D and port C. The use model for the base architecture is given below:

- All single cycle non-dependent accesses (read/write requests) to data that are in the BRAM use port A.

- The read accesses to data in BRAM that are consumers (means they are dependent on a certain write(s) in another thread) are done via port C.
- The write accesses to data by producers use port D.
- Port B is used for accesses to data that are either independent of those done via port C and/or port D or other non-time critical accesses.

In our experiments we have not used port B. In addition to the arbitration between ports, another important aspect is the dependency list. This is a list that contains a list of producers with data in this BRAM. This list is populated at configuration time since they are determined at design time using static analysis. Each entry in the list has two parts. The first part contains a dependency number, which is the number of threads that are dependent on this producer. This is used to count the number of consumer reads following each producer write, to determine completion of a produce-consume cycle and hence ending of the need for the address to be guarded. The second part of the entry is the base address of the data structure in BRAM. This is the address that consumer threads will provide to read the data. For multiple producer-consumer dependencies on a single address, we store the associated dependency number in each producer thread, which writes to the list using port D. Thus there are three different cases for access now:

- There is a read access on port C and if this address is present in list with a dependency number greater than zero, the access will be granted else it will block until a write happens that is related to this consumer
- A write on port D is allowed if there is a corresponding entry in the dependency list with a dependency number greater than zero.
- A read or write on port B is allowed as long as there are no current requests on port C or D. A blocking read request on port C is treated as a waiting request and can be overridden.

A content addressable memory (CAM) like structure is used for performing comparisons on all the addresses in the dependency list.

Thus at a meta-level the memory ports are being used as if one port is for general accesses as in any standard memory. The second port is providing a guarded access for those accesses that are dependent. Thus the second port is implementing the guard condition(s) that the user would have to write to implement such dependencies. In this particular technique the semantics followed are that of an arbitrated (bus-style) access to port C that introduces dynamic scheduling since one can add new consumers on this port without altering the base architecture. The latter aspect also introduces non-deterministic timing for cases where more than one producer-consumer pairs are mapped to the same BRAM structure. This is because the read accesses on port C are arbitrated as on a bus. Thus the arbitration scheme will determine the particular delay once the write happens. In our experiments we have implemented a simple round robin arbitration scheme. For our application domain of packet processing, the writes happen when packets arrive from a network and are probabilistic in nature.

We will now investigate another memory organization that eliminates part of the non-determinism, namely the delay on read operations once the producer performs a write operation.

3.2 Event-driven statically scheduled memory organization

We term the second memory organization as event-driven statically scheduled. Figure 3 shows the block diagram of an instance of such a memory organization. Unlike in the first organization, in this case we use a much more static interconnection organization. In this case, the first port (labeled A) behaves similarly to the earlier memory organization, namely for any generic (single cycle) access, and needs to be scheduled accordingly. The second port (labeled B) is connected to a more elaborate network and is reserved for inter-thread shared memory accesses. These accesses are the producer-consumer pairs that we have identified earlier.

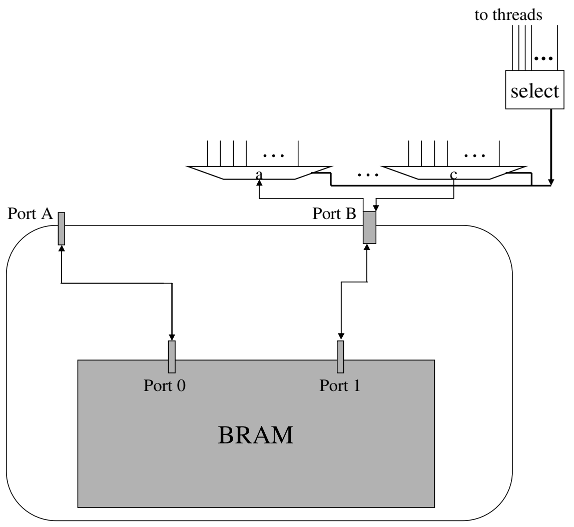

Figure 3 Event-driven statically scheduled memory organization

In this organization, a set of input and output signals related to reading and writing data from BRAM are routed via multiplexers (labeled c) and de-multiplexers (labeled a) that in turn form the required signals for port 1 on the BRAM. These multiplexers are driven by selection logic which is generated based on the producer-consumer relationships that we have obtained from the hic description. The selection logic uses modulo scheduling method to schedule the producer and consumer memory accesses. Modulo scheduling happens at two levels: between different producers and between different consumers of a given producer. Thus, at any given instant, modulo scheduling is in effect between a set of producers. Once a producer performs a write operation, modulo scheduling takes effect between different consumers of this producer. This scheduling however is implemented as an event from the producer thread into the first consumer thread, from the first consumer thread into the second, and so on.

For example, in the case of the example in Figure 1, first the selection will enable access to thread t1 only. Once the write related to x1 happens, then the corresponding reads for y1 and z1 will happen, in that order. The order in which y1 and z1 happen is determined at compile time. This in turn determines the nature of interconnection and is thus quite static. In addition, to ensure validity of operations, the consumer read accesses are initiated only when the selection logic generates the corresponding slot number. Thus the value output from selection logic is used as an event into the consumer threads. Based on this event the consumer threads perform read accesses. The producer thread starts the selection logic – until this point the selection logic is blocking, as required by our programming model. Thus the write by a producer is treated as an event by the consumers to which they respond. The order in which the consumer threads receive these events is statically scheduled. Thus we have accurate timing information once the write from the producer thread occurs. Note however that this does not mitigate the meta-level timing estimation problem related to the producer thread since that is a function of the input network traffic pattern.

The main advantage of this method is the better estimation of timing information. However if one needs to add new consumer threads, we have to modify both the multiplexing structure (as in arbitrated memory organization) as well as the state machine related to the thread since the producer event handlers need to be implemented. Use of FPGA programmability enables the latter aspect. We will now quantify the area overhead for both the memory organizations.

4. Preliminary Results and Analysis

To test the feasibility of our approach, we generated some simple designs that exercised both the memory controllers. First, we implemented the baseline architecture for the arbitrated memory organization. On top of this baseline architecture, we have mapped three different scenarios based on a simple Internet Protocol (IP) packet forwarding application. The three different scenarios scale the number of pseudo-ports that get mapped on to the read port (port C) and in turn impact the overhead in area and latency due to arbitration and scheduling. Second, we have implemented similar application scenarios for the event-driven statically scheduled memory organization. The three different scenarios comprise mapping two, four, and eight, pseudo-ports representing varying number of consumers for a single producer on to the consumer read port (i.e. port C for the first memory organization) for both the memory organizations respectively. Thus we have a single BRAM memory with different number of threads as consumers and a single thread as a producer.

In these experiments we used the Xilinx Virtex-II Pro as the target FPGA device. In particular we used a XC2VP20 part and the Xilinx ISE 6.3 (SP3) for the synthesis, placement, and routing, steps. The area and timing estimates are after placement and routing. The two-port IP forwarding application from which we derived the following cases used a total of 5430 slices, of which around 1000 slices were for the core forwarding function. The numbers presented here are the addition made to this total slice count.

Table 1 Required area for arbitrated memory organization

P/C	LUT	FF	Slices
1/2	145	66	92
1/4	231	66	135
1/8	319	66	179

Table 1 lists the different area figures obtained for the three cases stated above. The first column represents the number of producers and number of consumers (the second row has 1 producer and 2 consumers, and so on), the second column represents the number of lookup tables used on the FPGA fabric, the third column represents the number of flip-flops used, and the last column represents the total number of slices used by the memory organization. Note that these numbers are the overhead per-BRAM and are measured for a single BRAM. For each case, 125 MHz was the target clock rate. We achieved timing of 125.6 MHz, 130 MHz, and 158 MHz for the 8, 4, and 2 consumer thread cases respectively. The constant flip-flop count is due to the baseline architecture (as in Figure 2) which requires 66 flip-flops. The additional multiplexing of pseudo-ports does not contribute to the flip-flop count but only to the LUT count (and hence slice count).

Table 2 Required area for event-driven statically scheduled memory organization

P/C	LUT	FF	Slices
1/2	94	4	49
1/4	143	7	74
1/8	292	13	149

Table 2 lists the different area figures using similar conventions as above for the event-driven statically scheduled memory organization. In this scheme too, for all the three cases the synthesis, place and route was done unconstrained and we achieved timing of 129 MHz, 136 MHz, and 177 MHz for 8, 4, and 2 consumer thread cases.

As stated above, the total amount of area devoted to the core functionality of the IP forwarding is about 1000 slices. Thus depending upon the partitioning (of threads) and complexity of the functions the area overhead can vary from 5-20%. Hence this overhead needs to be considered *a priori* in the design partitioning process.

In addition, the latency of consumer read accesses once the corresponding producer write happens is not deterministic for the arbitrated memory organization. However, the arbitrated memory organization is simpler to implement since the base architecture is fixed and only the multiplexing required to support new consumer thread needs to be added and no changes need to be made to the thread related state machine(s). Thus for designs where there is enough slack in timing and a need to scale up in the future, the arbitrated memory organization is useful. For designs where timing is critical and needs more optimization, the event-driven memory organization is useful. In our design methodology we envisage providing the user with access to either of these implementations based on design time implementation constraints and parameters.

5. Related Work

There are three domains of prior art for the work presented in this paper: FPGA design tools, memory design and optimization, and programming languages.

In the domain of system-level FPGA and reconfigurable systems design tools, there has been prior work on automatically synthesizing memory organization for loop-oriented descriptions [10,11]. There has been significant prior art in the design automation community for generating memory architecture and related optimizations [8,9,10,11]. In both these domains, the majority of the effort has focused on estimation of memory size and, based on this estimation, allocation and scheduling (optimization) of memory accesses for a particular cost function.

Our work complements existing work in that we start with a concurrent specification and provide the user with different implementation paths to quickly generate a functionally correct RTL description by synthesizing inter-thread communication via shared memory utilizing existing memory size estimation and access pattern optimization techniques. In the programming languages community, software transaction memories (STM) have been proposed [2,3]. Complementary to STM, hardware transactional memories have also been researched [1]. We have discussed the main differences with them earlier in the introduction.

6. Conclusions and Future Work

In this paper, we have presented two techniques that allow users to implement inter-thread communication using a shared memory abstraction. We have illustrated the mapping process starting from a domain specific concurrent description and going onto a FPGA fabric utilizing the dedicated on-chip memory resources. We have quantified the overhead of implementing both these memory organizations as well as examining the benefits of each one. We have not yet investigated the impact of large amount of data dependencies on the size of list in arbitrated memory organization and this is part of current research. In addition, we are investigating the advantages of the arbitrated memory organization for reusability with respect to existing code (given the behavior analogous to that of bus-based systems).

7. REFERENCES

[1] M. Herlihy, J.E.B. Moss, "Transactional memory: architectural support for lock-free data structures," in Proc. 20th Annual Symposium on Computer Architecture (1993), ACM Press, pp. 289-300.

[2] N. Shavit, D. Touitou, "Software transactional memory," in Proc. 14th Annual ACM Symposium on Principles of distributed computing (1995), ACM Press, pp. 204-213.

[3] Tim Harris, Simon Marlow, et. al., "Composable memory transactions," in Proc. of ACM Symposium on Principles and Practice of Parallel Programming, June 2005.

[4] Virtex™-II Pro Platform FPGA Handbook (v1.0), Xilinx, January 2002.

[5] G. Brebner, "Multi-threading for logic-centric systems," in Proc. of 12th International Symposium on Field-Programmable Logic and Applications (FPL 2002), LNCS 2438, Springer-Verlag.

[6] D.W.Knapp, "Behavioral synthesis," Kluwer Academic Publishers, 1996.

[7] A. Aho, R. Sethi, J. Ullman, "Compilers," Addison Wesley, 1986.

[8] P. Panda, F. Catthoor, et. al., "Data and memory optimization techniques for embedded systems,' ACM Transactions on Design Automation of Electronic Systems, 6(2), April 2001.

[9] F. Catthoor, S. Wuytack, et. al., "Custom memory management methodology: Exploration of memory organisation for embedded multimedia system design," Kluwer Academic Pulishers, October 1998.

[10] M. Weinhardt and W. Luk, "Memory access optimization and RAM inference for pipeline vectorization," in Proc. of 9th International Workshop on Field-Programmable Logic and Applications FPL'99, LNCS 1673, Springer-Verlag.

[11] G. Venkataramani, W. Najjar, et. al., "Automatic compilation to a coarse-grained reconfigurable system-on-chip," in Proc. of ACM transactions on Embedded Computing Systems, Vol. 2, Issue 4, November 2003, pp. 560-589.