Appears in *IEEE International High Level Design Validation and Test Workshop*, November 2007

Transactors for Parallel Hardware and Software Co-Design

Krste Asanović
Computer Science Division
University of California at Berkeley
krste@eecs.berkeley.edu

1 Introduction

Complex, high-performance, low-power information processing systems usually incorporate a mixture of hardware and software elements, and pose significant design challenges. Conventional register-transfer level (RTL) hardware design methodologies are too low-level, requiring designs to be partitioned into collections of combinational gates separated by clocked registers. Conversely, threaded parallel software design methodologies provide a very high-level specification from which it is difficult to synthesize efficient gate-level implementations. In this paper, we introduce the *transactor* (*trans*actional *actor*) model to provide a natural level of design representation for both hardware and software implementation. As shown in Fig. 1, a design is modeled as a network of communicating transactors connected by message queues, where each transactor performs atomic actions and sends output messages based on its local state and messages arriving on its input queues.

Transactors support a higher-level design style, we term UTL (Unit-Transaction Level) design. Transaction-level modeling has become a popular starting point for modern embedded system design, but the goal of transactors is to support a complete transaction-level *synthesis* flow into hardware or software. A transactor can be implemented as custom circuitry, or as a gate-level netlist mapped to standard cells or an FPGA. A transactor can also be implemented as software running on a conventional instruction-set processor.

2 The Transactor Model

The transactor computation model is based on guarded atomic actions [2, 7], to cleanly specify non-determinate concurrent programs.

A system is described as a hierarchical composition of *transactors*. A transactor unit has four components: architectural state; buffered input and output channels that provide decoupled connections between transactors; a set of atomic transactions that can read from the input channels, mutate private state, and write to the output channels; and a scheduler that selects the next transaction to perform (Fig. 2).

The *architectural state* of a transactor is the state that persists between transactions, and hence which can affect the operation of future transactions or the scheduler. Architectural state is of fixed size. An implementation of a transactor may contain further microarchitectural state (e.g. pipeline registers or caches), but this is never visible from the transactor's interface (i.e., it can never change the behavior of any transaction).

Input and output channels provide buffered point-to-point connections between units. Channels are unidirectional with a single sender and a single receiver, and carry messages with a fixed maximum size.

A *transaction* describes a possible computation to be performed. Transactions are guarded atomic actions, where the guard is a predicate over the state of the head of each input channel read, the status of the tail of each output channel potentially written to, and the unit's architectural state. Transactions execute atomically with respect to each other. A transaction can read at most one message from each input channel, and when a transaction fires, any messages it read are removed from the input channels. A transaction can write at most one message to each output channel.

The *scheduler* contains a combinational function which

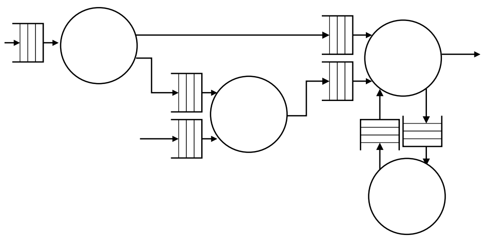
Figure 1: UTL design containing transactor units communicating over message queues.

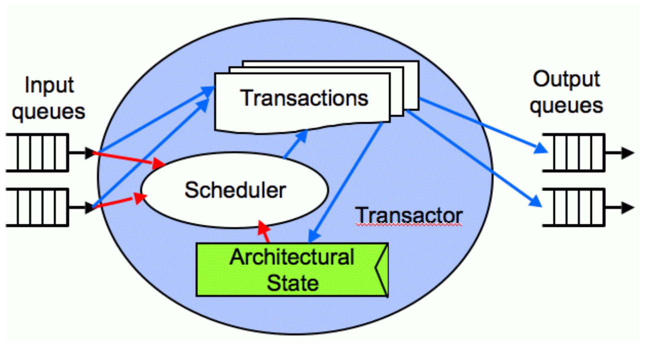

Figure 2: Anatomy of a transactor.

selects the next transaction to be performed, if any. The scheduling function can read architectural state, the empty/full state of each input queue, and the message at the head of each input queue. The scheduler can not update state unless a transaction is fired, but can then update scheduler state to implement various scheduling policies. The scheduler functionality can in principle be merged into transactions (by folding scheduler priority equations into transaction guards, and by folding scheduler state updates into transaction state updates), but the separation makes code more modular and easier to read.

The operational semantics of a collection of transactors are as follows: Non-deterministically select a transaction that is ready to execute on some unit, execute that transaction, and repeat. If no transaction can execute, the system is either finished, waiting on external input to arrive, waiting for external output to drain, or deadlocked.

3 Router Example

Fig. 3 shows a pseudo-code description of a transactor that routes two input streams of message packets to two output streams. The two transactions are `route`, which moves packets from an input to one of the two outputs, and `route_kill`, which removes malformed packets from the system while incrementing a `bad_packet` counter. The `routable` function determines whether the input is a valid packet, and the `route_func` function determines which of the two output streams the packet should leave on if routable.

The transactor contains two pieces of architectural state, the bad packet counter plus a one-bit value, `last`, used to ensure a fair schedule. The scheduler state is considered part of the architectural state because if affects the externally visible behavior of the unit.

The scheduler has two components: a combinational scheduling function that prioritizes and selects the next transaction among all ready transactions, and a scheduler state update operation that only occurs after a transaction has been fired. The scheduling function is restricted to be combinational, so that no state is updated except when a transaction fires. The scheduler state update is folded into the same atomic action as the transaction that was fired.

By default, transactions listed sequentially within a `schedule` statement have decreasing priority. Schedules can be nested, and can take user-supplied priority modifier routines to provide more complex dynamic schedules. The example uses a standard priority modifier routine, (`round_robin`), which implements one form of fair round-robin scheduling using the scheduler state, `last`, to remember the index of the last transaction to fire. The priority of the constituent transactions is modified to give the lowest priority to the transaction that just fired, with priority increasing as the numeric label decreases.

The `reset` transaction, which depends only on the implicit `reset` channel, always has the highest priority and is used to bring the unit into a known state before it begins processing messages. The reset channel is the only example of synchronous inter-unit communication in the transactor network, as each unit must simultaneously receive a reset token.

```
transaction route(
  input pkt in,
  output pkt out0,
  output pkt out1)
{
  when (routable(in))
    if (route_func(in)==0)
       out0 = in;
    else
       out1 = in;
}

transaction route_kill(
  input pkt in,
  int[32]& bad_packets)
{
  when (!routable(in))
    bad_packets++;
}

transactor
router(input pkt in0,
       input pkt in1,
       output pkt out0,
       output pkt out1)
{
  int[32] bad_packets;
  int[1] last; // Fair scheduler state.
  schedule {
    reset { bad_packets = 0; last = 0; };
    route_kill(in0, bad_packets);
    route_kill(in1, bad_packets);
    schedule round_robin(last) {
      (0): route(in0, out0, out1);
      (1): route(in1, out0, out1);
};};}
```

Figure 3: Routing unit example.

4 Transactors in Hardware

Hardware designs can follow a template consisting of a datapath to execute the transactional code, together with an arbiter implementing the scheduling function. The execution datapath can be pipelined, and a single transaction might require several passes through the pipeline.

The transactor description is intended to enable a wide range of efficient hardware implementations. For example, one implementation of the `router` function given above could pipeline its execution, and allow only a single `route` transaction into the pipeline each cycle. Alternatively, another design might be unpipelined but allow the two `route` transactions to fire on the same cycle provided they were going to write to different outputs. Similarly, a more sophisticated implementation could allow both `route_kill` transactions to fire on the same cycle, updating the shared `bad_packets` state by 0, 1, or 2.

Transactors are intended to represent relatively coarse-grain units of functionality, perhaps 10,000–100,000 gates of hardware. These units are small enough that conventional RTL design tools perform adequately within a unit [9]. A large sub-100 nm chip design might contain many thousands of hardware transactors, and it is at this global scale that a UTL discipline such as transactors can help with logical and physical design complexity:

- Transactors encode locality of access, separating local computation from global communication.
- The message queues decouple the state machines of communicating transactors, limiting combinational paths in control logic.
- Transactors use a latency-insensitive design style [5], where functionality should not depend on communication latency. This enables more flexibility in physical design choices, such as the use of pipelined or multiplexed global wires, or a GALS clocking strategy.
- Transactions provide a natural granularity for implementing soft error correction through check and retry.

5 Transactors in Software

Transactors are easily mapped to software implementations. The architectural state becomes a data structure in memory. Each transaction can be mapped to a serial thread of code that executes to completion once fired. The scheduler becomes a separate piece of code that is run sequentially after each transaction completes. If no transactions are ready to fire, the whole transactor can be descheduled. The scheduler need only run again to check guard conditions if new inputs arrive or the output channels drain. The efficiency of invoking the scheduler code on communication events will depend on the underlying hardware implementation. When multiple transactor instances are mapped to the same processor, their scheduling functions can be merged to improve efficiency.

6 Related Work

The transactor model builds upon a number of earlier approaches.

Kahn networks [8] provide decoupled communication between actors but do not provide for non-deterministic arrival of input messages. Also, execution within an actor is sequential whereas transactors allow concurrent actions internal to an actor. Kahn processes of bounded size can be mapped to transactors, where input or output events terminate transactions (i.e., code between I/O events can be represented by one or more transactions).

The CSP computational model [6] underlies the Occam language [3]. Parallel processes communicate and synchronize via a rendezvous over a communication channel. Rendezvous exposes round-trip communication latency in implementations, and many Occam programs required additional explicit buffer processes to decouple units and tolerate long and variable latency communication. Also, CSP does not provide a mechanism to manage mutable state shared by concurrent processes.

The Cal Actor Language (CAL) [4] shares many similarities to the transactor model, however, CAL is less well defined as the execution semantics mostly depend on an external environment.

The use of guarded atomic commands in TRS [7] and Bluespec [1] inspired much of the thinking behind transactors. The transactor model adds the UTL design discipline of units decoupled with message queues, and also allows for multi-cycle transactions possibly of data-dependent duration (i.e., operations that cannot be completely unfolded in space in a hardware implementation).

7 Summary

The use of higher-level design specifications is required for large scale embedded systems, yet these must admit efficient hardware and software implementations. The transactor model separates local computation from global communication, and avoids overspecifying the execution of computations within each unit. The use of guarded atomic commands provides a clean model for concurrent activities that share state within each unit, and supports computations on non-deterministic input streams.

References

[1] Bluespec Inc. Bluespec(tm) SystemVerilog Reference Guide: Description of the Bluespec SystemVerilog Language and Libraries, Waltham, MA, 2004.

[2] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[3] Inmos Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[4] J. Eker and J. W. Janneck. CAL language report. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003.

[5] L.P. Carloni et al. A methodology for correct-by-construction latency-insensitive design. In *Proc. ICCAD*, November 1999.

[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[7] J. Hoe and Arvind. Operation-centric hardware descriptions and synthesis. In *IEEE TCAD*, 2004.

[8] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475, August 1974.

[9] D. Sylvester and K. Keutzer. Impact of small process geometries on microarchitectures in systems on a chip. In *Proc. IEEE*, volume 89, April 2001.