xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification

Satrajit Chatterjee, Michael Kishinevsky, and Umit Y. Ogras
Intel Corporation

Abstract

Although communication fabrics at the microarchitectural level are mainly composed of standard primitives such as queues and arbiters, to get an executable model one has to connect these primitives with glue logic to complete the description. In this paper we identify a richer set of microarchitectural primitives that allows us to describe complete systems by composition alone. This enables us to build models faster (since models are now simply wiring diagrams at an appropriate level of abstraction) and to avoid common modeling errors such as inadvertent loss of data due to incorrect timing assumptions. Our models are formal and they are used for model checking as well as dynamic validation and performance modeling. However, unlike other formalisms this approach leads to a precise yet intuitive graphical notation for microarchitecture that captures timing and functionality in sufficient detail to be useful for reasoning about correctness and for communicating microarchitectural ideas to RTL and circuit designers and validators.

1 Introduction

Time-to-market is one of the major constraints of system-on-chip (SoC) designs. Project teams usually rely on standard IPs re-used across different products to achieve quick turn around time. Therefore, efficient system-level interconnect solutions that can seamlessly glue different IPs become an integral component of SoC designs. However, if not designed carefully, deadlocks and livelocks in system-level interconnect can present a significant challenges to quick SoC integration. Distributed nature of the system-level interconnect and tricky interaction between many IPs make this problem hard to understand and debug. This is in strong contrast to other functional problems which often may be narrowed down to a few blocks in the design that are well understood both by the implementers of the block as well as the system architects. Furthermore, deadlock problems at the system level are hard to fix after they are found: To fix them cheaply requires sacrificing performance by limiting concurrency and to fix them properly may require altering many blocks.

Given the shortcomings of RTL and performance models for system-level deadlock analysis, we propose an approach based on creating executable and

analyzable high-level models to reason about deadlocks. We address two things that make this approach to pre-silicon deadlock validation challenging in practice. (1) Productivity and resourcing problem: developing models with the realistic levels of details sufficient for reasoning about deadlocks and livelocks is time consuming and requires highly trained validators; and (2) Verification problem: practical methods for formal verification of liveness properties are not sufficiently efficient to handle realistic models. In this paper we discuss an approach that attempts to solve both of the above problems and will illustrate it on several examples. We call these models xMAS for eXecutable Microarchitectural Specification. Given the close correspondence of the diagrams and the models we use the terms xMAS model and xMAS diagram interchangeably.

To address the productivity problem we identify a set of microarchitectural primitives that allows us to describe complete systems by composition alone without any intermediate glue logic. This enables us to build models faster (since models are now simply wiring diagrams at an appropriate level of abstraction) and to avoid common modeling errors such as inadvertent loss of data due to incorrect timing assumptions. Furthermore, one does not have to think about the subtle control and timing logic in these highly concurrent systems. Since our models are built from a structural composition of these primitive blocks, it is natural to represent these models diagrammatically. This leads to a schematic view of the microarchitecture (similar to schematics for circuits). Unlike the normal box diagrams seen in microarchitectural documents which have no semantics, these diagrams have precise semantics that capture both timing and functionality.

To address the verification problem we generate abstract Verilog out of our models and use in-house and academic verification tools for formal bug-hunting by reducing liveness problem to standard safety properties. This is possible since our models are formal and may be used for model checking as well as dynamic validation and performance modeling. In addition we automatically generate inductive invariants of the systems that are comprised from local invariants of our well defined primitives and global invariants (such as preserving the number of credits in virtual channels) that are extracted through formal flow analysis of the system. Generating these invariants allows us to do quick formal proofs of non-trivial system properties.

Timing model. We use a synchronous model of time [1]. Each primitive is defined by a set of equations that can be trivially translated into synthesizable Verilog with a single clock and edge-triggered flip-flops. Giving asynchronous semantics to our primitives is equally possible, but is out of scope of this paper.

2 Overview of XMAS Modeling Approach

2.1 Basic Communication

Consider a synchronous FIFO queue with a standard interface comprising a read port and a write port as shown in Figure 1 (a). The queue has two parameters: size k (the number of elements it can contain) and a type τ (the type of elements

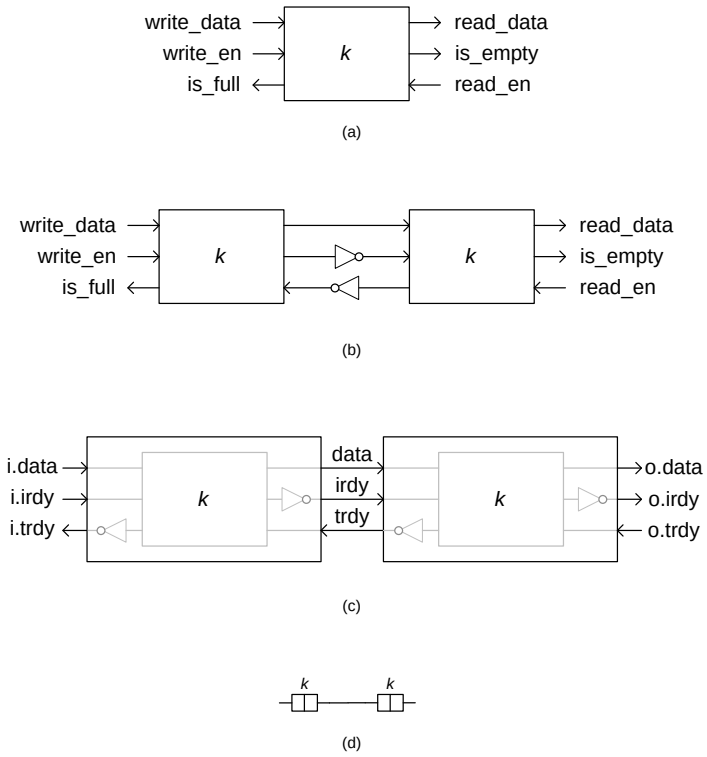

Figure 1: (a) A synchronous FIFO queue with a write port (on the left) and a read port (on the right). The queue can store k data elements. In each clock cycle, if the queue is not full, a new element may be inserted; and if the queue is not empty, the oldest element may be removed. read_data exposes the oldest element if the queue is not empty. There is no bypass: if the queue is empty, the incoming data appears at the output one cycle later. (b) Composing two queues requires additional glue logic. (c) By choosing an appropriate interface for the queue, it is possible to compose two queues without additional glue logic. (d) We can abbreviate the preceeding figure by using a single line to represent the channel connecting the two queues instead of separate lines for the data, irdy and trdy wires of the channel.

it can contain). To compose two instances of such a queue back-to-back, we need some glue logic as shown in Figure 1 (b) to ensure that data elements are transferred "correctly" from queue A to queue B i.e. a transfer happens if and only if queue A is not empty and queue B is not full.

Looking at Figure 1 (b) it may be obvious how to redefine the queue interface of Figure 1 (a) so that no additional glue logic is necessary. For instance, consider the following interface definition in terms of a read port o (output of the queue) and a write port i (input to the queue):

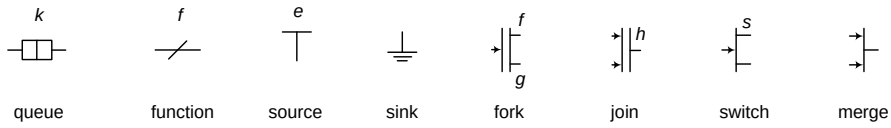

Figure 2: A key showing the symbols for the various primitives used to model microarchitectural blocks. The italicized letters (k, f, e, g, h and s) indicate parameters. Whenever we use these primitives in a diagram we need to specify values for these parameters. Often, to avoid clutter we do not show these values explicitly trusting that they are clear from the context. For some components such as the fork, we place a parameter close to the "corresponding" port in the diagram.

o.data := read_data	write_data := i.data
o.irdy := **not** is_empty	write_en := i.irdy
i.trdy := **not** is_full	read_en := o.trdy

where we define the new interface in terms of the old one by folding the inverters in the glue logic of Figure 1 (b) into the queue itself.

With this new interface, one can directly connect two queues structurally without requiring any additional glue logic as shown in Figure 1 (c). This is the central idea of this paper: by defining a library of elements that adhere to a standard interface, we wish to describe interesting microarchitectures by simple structural connections.

Note that the two queues are connected by three wires: data, irdy (for initiator ready) and trdy (for target ready). We call this triple of wires a *channel*. Channels are the only communication mechanism in our framework. As in the queue example, a channel always connects two ports: an *initiator* and a *target*. The data and irdy wires go from the initiator to the target, whereas the trdy wire goes from target to initiator. irdy indicates that the initiator is ready to send data, and trdy indicates that the target is ready to accept data. Data are transferred exactly on those clock cycles when both irdy and trdy are true. Each channel has a type τ which indicates the type of data it carries. Channels induce types on the ports they connect to. For example, both i and o ports of a queue have type τ and this is denoted by $i, o : \tau$.

Diagrammatically, rather than a line for each of the three wires in a channel, we show only a single line (for a channel). Thus the Figure 1 (c) can be abbreviated to Figure 1 (d). Note that with the queue interfaces as described above and given a queue implementation, Figure 1 (d) is a very precise description of a system that specifies both functionality and timing. In particular this description is precise enough for model checking or generating synthesizable Verilog.

2.2 xMAS Primitives

The xMAS primitives are listed in Figure 2. Each primitive has a unique graphical notation and well-defined formal semantics. This enables sketching a variety of micro-architectural models without any ambiguity in the functionality.

The first primitive in Figure 2 is the queue which has been described in the previous section. The next primitive is the *function* primitive which takes a combinational function f as a parameter, and applies f to an incoming packet to produce the output packet. The *source* and *sink* primitives create and consume packets non-deterministically. The source is parameterized by an expression e that defines the type and content of the packets that the source creates. The *switch* primitive is used to route incoming packets to one or the other output (as decided by the select function s which is a parameter). The *merge* primitive is used to model (fair) arbitration. In any given cycle it allows a packet from at most one of its inputs to be sent to the output and applies back-pressure to the other inputs.[1] Finally, the *fork* and *join* primitives are used for synchronization. They are described below in more detail for illustration. Similar detailed definitions of the other primitives can be found in [11].

A fork has one input port $i : \alpha$ and two output ports $a : \beta$ and $b : \gamma$ parameterized by two functions $f : \alpha \rightarrow \beta$ and $g : \alpha \rightarrow \gamma$. Intuitively, a fork takes an input packet and creates a packet at each output. It coordinates the input and outputs so that a transfer only takes place when the input is ready to send and both the outputs are ready to receive. Formally,

a.irdy := i.irdy **and** b.trdy a.data := f(i.data)
b.irdy := i.irdy **and** a.trdy b.data := g(i.data)
i.trdy := a.trdy **and** b.trdy

A join is the dual of fork. It has two input ports $a : \alpha$ and $b : \beta$ and one output port $o : \gamma$. It is parameterized by a single function $h : \alpha \times \beta \rightarrow \gamma$. Intuitively, a join takes two input packets (one at each input) and produces a single output packet. It coordinates the inputs and output so that a transfer only takes place when the inputs are ready to send and the output is ready to receive. Formally,

a.trdy := o.trdy **and** b.irdy
b.trdy := o.trdy **and** a.irdy
o.irdy := a.irdy **and** b.irdy o.data := h(a.data, b.data)

Note the duality of the join equations with the fork equations for the irdy and trdy signals.

2.3 xMAS Macros

There are several microarchitectural blocks that are common to many models. Although these blocks are made from the primitives described in Section 2 it is convenient to abbreviate them with special symbols and we call these macros.

Figure 3 shows three macros. The first two of them are used in this paper. Figure 3 (a) shows the macro symbol for non-deterministic delay and its definition in terms of the xMAS primitives. Figure 3 (b) shows the macro symbol for a credit counter and its definition. Credits are a common microarchitectural design pattern for flow control and resource allocation in distributed systems [4] and we show how they are used in a larger example in the next section. Figure 3

[1] Although only two inputs are shown in Figure 2 a merge can have multiple inputs.

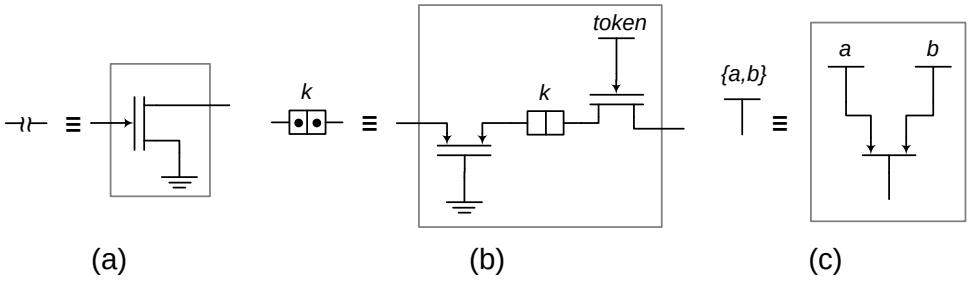

Figure 3: Three commonly used macros and their implementation using xMAS primitives: a non-deterministic delay (a), a credit counter (b), and a multiplexed source (c). The functions of the forks and joins are identity. The sink in the credit counter is eager i.e. it can consume a packet every cycle.

(c) shows a multiplexed source that non-deterministically injects one of the two values and its definition using simple sources injecting one value each. Note that any reused block, such as an agent in Figure 4 can be viewed as a macro.

In this scheme credits are modeled by the unit type, i.e. an enumerated type with only one possible value which we shall call *token*. The credit counter macro ensures that at most k outstanding tokens are sent out at any given time. The queue in the credit logic contains as many tokens as there are outstanding credits issued and applies back-pressure to the token source when it is full. On the other hand, the sink in the credit logic is eager and applies no back-pressure. Hence, the join in the credit logic ensures that a new input will be accepted, if the credit queue has a token. Thus in the context of credits, the credit counter macro behaves as though it were a queue with k tokens at initialization.

3 Two Illustrative Examples

The primitives presented in Section 2 can be used to model a wide variety of microarchitectural examples. In this section we present two non-trivial examples to give you a sense of the modeling power of these primitives. More examples and a more leisurely treatment can be found in [11].

3.1 A Simple Fabric with Two Master-Target Agents

Figure 4 shows two agents P and Q communicating over a trivial fabric composed of six queues. Packets are modeled by an enumerated type that has two values: *req* (request) and *rsp* (response). Each agent creates new requests for the other agent. When an agent receives a request, it produces a response after a non-deterministic delay (using the non-deterministic delay macro of Section 2.3). The response is sent back to the original agent where it is sunk when the sink is ready to receive it. Thus each agent behaves like a master that produces requests and responses and a target that consumes responses and requests.

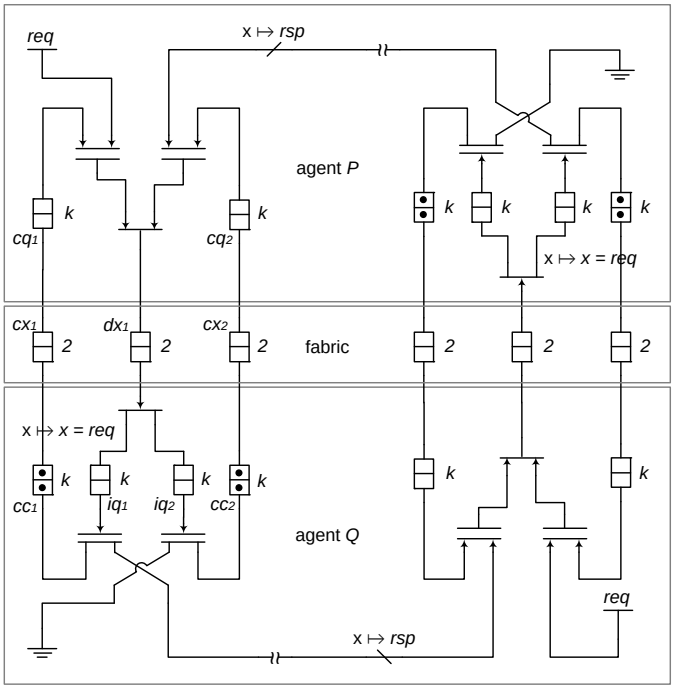

Figure 4: Example showing a pair of agents communicating over a simple fabric (see text for details). Since each symbol has a precise formal semantics (see Section 2) this figure is a precise executable description.

Communication between agents is done through the virtual channels. Consider agent P as example. It sends requests and responses to agent Q through the shared channel r and the data transfer queue dx_1 and then to two ingress queues iq_1 and iq_2, one per message type. At the output of P, an arbiter modeled by the merge primitive selects fairly between *req* and *rsp* messages that are exposed to arbitration only if they have credit tokens inside the corresponding credit queues cq_1 and cq_2. Credits are initialized inside the credit counters cc_1 and cc_2 (in agent Q) to the values equal to the sizes of the ingress queues iq_1 and iq_2, i.e. to k (see Section 2.3). Credits are sent to agents through fabric credit queues e.g. through queue cx_1 for req credits for agent P.

Discussion. How does such a model compare with one written directly in a language such as Verilog? First, typically one would have a valid wire for r (corresponding to our irdy), but would not have the trdy (since there is no back-pressure on r since the credit mechanism ensures that there is always room in the ingress queues of Q. (For this reason channel r is called *non-blocking*.) However, by explicitly modeling the trdy wire, it becomes easy to add assertions to validate our microarchitectural timing assumptions. Furthermore, it is clear from the diagram that for r to be non-blocking, all the channels downstream all

the way up to the output of queues iq_1 and iq_2 must be non-blocking. Thus from xMAS diagrams it is possible to reason about the propagation of microarchitectural timing constraints (and even channel utilizations). Aside: if the output of a queue is non-blocking, then the queue can be simplified to a flip-flop since it contains at most one element at any time. For this reason we do not need a special flop-like primitive to model delays.

Second, it may appear odd and wasteful to use queues to model counters to track credit. It is not wasteful: one can trivially detect queues whose type is the unit type and optimize them to counters. However, by modeling credits in this manner, it is easy to ensure that credits don't get "lost." Furthermore, it is trivial to modify the model as the microarchitecture changes. For instance, if it is required that the credits for req and rsp use the same channel through the fabric, then one can easily modify the diagram by using an enumerated type with two values to model the credits and introducing a new merge primitive at the outputs of cc_1 and cc_2 to feed into a common path through the fabric, and a switch to separate the credits by their type to feed into cq_1 and cq_2. Also, from the diagram, it is easy to see that this merging of the credits would not lead to a throughput loss since the flow of credits would need only match the flow of data through r. This kind of reasoning and modification is much harder with a conventional model.

Third, the precise timing semantics of Figure 4 allow us to reason about performance issues. For example, because a credit has a round-trip time of 5 units (cc_1, cx_1, cq_1, dx_1, iq_1, and finally back to cc_1) it is clear that $k \geq 5$ to allow full utilization on r in the worst case where there is only one type of message (i.e. all requests or all responses).

Finally, the xMAS diagram makes explicit the dependencies in the system thus making it easier to reason about deadlocks. For example, suppose credits are incorrectly sized (i.e. are more than the size of the ingress queues), then it is easy to see that queues dx_1 and iq_1 on one side and the corresponding queues on the other side can fill-up with requests leading to a deadlock since neither agent is able to write a response as the fabric queues are filled with requests.

3.2 A Scoreboard for Tracking Transactions

Figure 5 shows how a two-entry scoreboard may be modeled using the primitives of Section 2. An incoming transaction on the left needs to obtain a tag before it can enter the scoreboard. Different tags are used to distinguish different in-flight transactions in the scoreboard. In this example, the scoreboard supports two simultaneous in-flight transactions and hence there are two tags: tagA and tagB. Once the transaction enters the scoreboard it competes with the other transaction (if there is one) to enter the first phase of processing. The results of this phase may return out of order: tags are used to match a result with the corresponding transaction in the scoreboard. Once the result of the first phase is returned, the transaction moves on to the second phase. After the second phase is done, the transaction becomes eligible for retirement. When it wins arbitration, it retires and releases its tag which is then recycled for use by a future transaction.

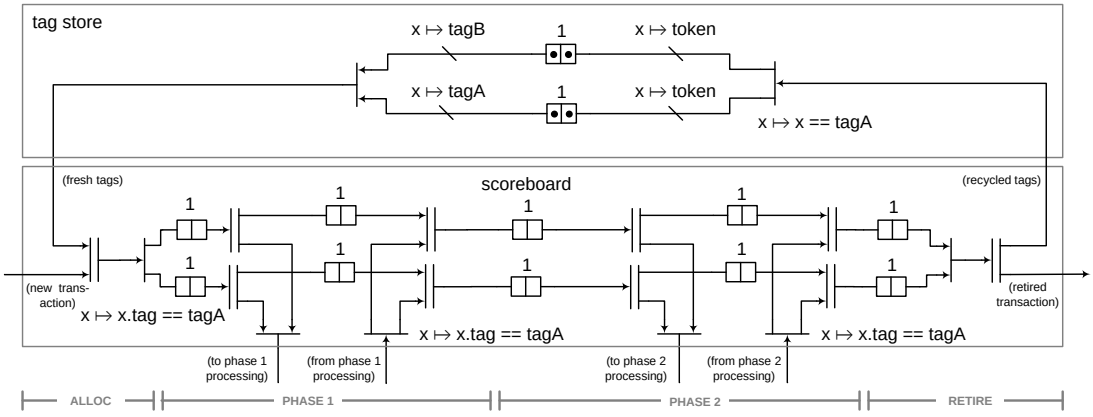

Figure 5: A fragment of an xMAS diagram showing how a simple scoreboard to support out-of-order processing may be modeled using the primitives of Section 2. This scoreboard can track two transactions in flight at a time. The transactions are tracked by their tags (tagA and tagB), and the two credit counters (with 1 credit each) in the tag store ensure that only one tagA and one tagB are in circulation. See Section 3.2 for details.

Discussion. This pattern may be used to model complicated control logic in a natural transactional style. For example, using this pattern one can model a multi-processor memory controller that issues snoops and collects replies (in phase 1) and updates memory (in phase 2).

It is easy to extend this example to track more than two simultaneous transactions in flight by increasing the set of tags, and to support transactions with multiple phases with complicated dependencies between them. Finally, note that even a big architectural change such as in-order retirement can be incorporated with little effort by adding a fork and a queue (to remember the orginal order) in the allocation phase and a join per tag in the retirement phase.

4 Practical experience

4.1 Tool Chain

We have built a system that provides a C++ API to describe xMAS diagrams. The user simply sketches the model and writes the xMAS model using this api. Modeling bugs are quickly found and corrected due to the automatically generated assertions (e.g. mutual exclusion of demux conditions) and build errors and warnings (e.g. dangling channels). Once the model type checks, the first bugs found are actual microarchitectural bugs such as an incorrect timing assumption (often detected by a failing non-blocking assertion) or a deadlock. We find that there are no subtle modeling bugs such as accidentally dropped or overwritten data.

From the C++ description we can generate output for several backends

Table 1: Experience on industrial examples. Effort is measured in person days. Since the models are parameterized and hierarchical, we roughly measure the design complexity by the number of unique instances. See text for more details.

Design	Effort	Complexity	Validation
Ring	2	42	Proof
SMF	2	47	Proof
IOF	5	194	BMC & dynamic
NC	10	≈ 200	Paper proof
TD	14	≈ 250	Paper proof

including synthesizable Verilog. This is used for performance analysis, dynamic validation, model checking and coverage analysis. Coverage targets are naturally formulated in terms of irdy and trdy signals and hence can be automatically generated.

4.2 Design Examples

We have successfully modeled and validated the microarchitecture of a number of complex industrial designs. Table 1 summarizes our experience. **Ring** is a model of a ring interconnect which was validated for absence of injection starvation and deadlock freedom. **SMF** is a model of a sideband messaging router (and agent environment) which was validated for deadlock freedom in the presence of self-traffic. **IOF** is a pipelined memory switch (and agent environment) allowing peer traffic that was validated for deadlock and starvation. The last two examples are highly concurrent with virtual channels, ordering restrictions, split-transactions and deeply pipelined architectures allowing tens of simultaneous in-flight transactions even in minimal configurations.

In all cases the time taken to build the model was significantly shorter than the other modeling efforts. As a reference point, a SystemC/TLM model of **IOF** without peer traffic took several person-months. Furthermore, our experience developing and maintaining various models of rings in SystemC show that it is relatively easy to introduce bugs due to inadvertent packet overwriting. By switching to a more restricted modeling style as proposed, we believe that such bugs can be avoided improving productivity.

The last two examples in Table 1 are models of much larger blocks and were modeled at a higher level of abstraction. **NC** is an SoC memory complex and **TD** is a controller in a distributed tag directory. Most of the modeling effort was spent in understanding the microarchitecture from hundreds of pages of documentation. The final model was compact enough to fit on a few sheets of paper. This proved to be indispensible in reasoning about deadlock freedom.

4.3 Verification Experience

We have found that bug hunting with bounded model checking works well but proofs are hard even for safety properties due to high degree of concurrency and

pipelining. Although these models are abstract, since they capture an entire system, they are hard for the model checking tools. For instance, to check agent injection liveness on **IOF** in minimal configuration (585 flipflops) took about 7 days on a 3 GHz Intel Xeon CPU for 25 frames of BMC. Here, the graphical notation has proven to be an indispensable fallback for pen-and-paper reasoning as well as for compositional proofs. Note that this is usually not an option with a more conventional model in say Verilog, Esterel, or Murphi.

Proof Generation. If an xMAS diagram satisfies certain properties, then an important class of safety and liveness properties can be proved automatically. This is done by leveraging the high-level structure available in the xMAS diagram to add additional invariants and to do propagation of deadlock equations. By adding these invariants, it is possible to make the entire set of properties 1-step inductive, thereby allowing RTL model checkers to prove them easily. For example, since all the queues in a system are easily identified, we can add invariants saying that if the output of a queue satisfies a particular property, then the input of the queue and all packets in the queue must also satisfy the property at all times. It is also possible to infer more global invariants such as those relating the occupancies of several queues in the system. By adding these invariants, the task of the model checker becomes much simpler often allowing it to find otherwise intractable proofs in seconds using induction. The details of proof generation with inductive invariants are presented in [3] and leveraging it for verification of liveness properties in [6].

5 Related Work

Modeling Frameworks. Our focus is on identifying a small set of primitives that is rich enough to model common microarchitectural design patterns used in practice; once the right primitives are identified, they can be expressed in any reasonable modeling framework. For concreteness we provide synchronous semantics for our components but they could be compiled down to Petri nets [12]; be modeled by dataflow actors [8]; in guarded command style [2] or be written as SystemC processes. In the context of any one of these frameworks, the proposed method may be viewed as simply a specific modeling style that promises greater productivity by minimizing the inadvertent modeling bugs. Thus from the same description one may generate different backend specifications allowing the strengths of the associated analysis and verification tools to be fully leveraged.

Performance Modeling. A number of mathematical and simulations models of communication fabrics for both power and performance are proposed in the networks-on-chip (NoC) literature (such as [10]), and processor performance modeling frameworks (such as ASIM [5]). A key difference is that unlike these simulators our models have a direct path to formal verification.

Formal Frameworks. There is a rich literature on specifying systems by a set of actions or transactions that modify shared state [2]; on compiling these specifications to RTL [7]; and on verifying RTL against transaction-level spec-

ifications [9]. In this context, our approach may perhaps be seen as a better way to describe RTL. For some systems such as communication fabrics our approach may form a basis for RTL generators starting from easy to specify formal models.

6 Conclusion

So far our focus has been on modeling and validating existing microarchitectures. The designs that are most naturally expressed in this notation are highly concurrent and distributed i.e. systems that are designed in a physically-aware manner. By thinking of new microarchitectures in terms of these primitives, it may be possible to obtain designs that admit more efficient circuit implementations compared to designs obtained through standard RTL design which have more centralized control.

References

[1] A. Benveniste et al. The Synchronous Language Twelve Years Later, in *Proc. of the IEEE*, 91(1):64-83, Jan 2003.

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

[3] S. Chatterjee and M. Kishinevsky. Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics, in *Proc. of Intl. Conf. on Computer Aided Verification*, 2010 (to appear).

[4] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.

[5] J. Emer *et al.* Asim: A Performance Model Framework, *IEEE Computer,* vol. 35, no. 2, pp. 68-76, February, 2002.

[6] A. Gotmanov and S. Chatterjee and M. Kishinevsky. Verifying deadlock-freedom of communication fabrics, In *Proc. VMCAI 2011*, pp. 214–231

[7] J.C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions, In *Proc. ICCAD 2000*, pp. 511–518

[8] E. Lee and T. Parks. Dataflow Process Networks, in *Proc. IEEE,* vol. 83, pp. 1-63, May 1995.

[9] Y. Mahajan et al. Verification driven Formal Architecture and Microarchitecture Modeling, in *MEMOCODE,* 2007.

[10] G. de Micheli and L. Benini. *Networks on Chips*, Morgan Kaufmann, 2006.

[11] S. Chatterjee, M. Kishinevsky and U. Y. Ogras. Quick Formal Modeling of Communication Fabrics to Enable Verification, in *Proc. IEEE,* HLDVT 2010.

[12] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541580, 1989.