Hunting deadlocks efficiently in microarchitectural models of communication fabrics

Freek Verbeek
Radboud University Nijmegen
Institute for Computing and Information Sciences
The Netherlands
Email: f.verbeek@cs.ru.nl

Julien Schmaltz
Open University of the Netherlands
School of Computer Science
The Netherlands
Email: julien.schmaltz@ou.nl

***Abstract*—Communication fabrics constitute an important challenge for the design and verification of multi-core architectures. To enable their formal analysis, microarchitectural models have been proposed as an efficient abstraction capturing the high-level structure of designs. We propose a novel algorithm to deadlock verification of microarchitectural designs. The basic idea of our algorithm is to capture the structure of the wait-for relations of a microarchitectural model in a *labelled waiting-graph* and to express a deadlock as a feasible *closed subgraph* of the waiting-graph. We apply our algorithm to academic and industrial Networks-on-Chip (NoC) designs. With examples we show that our tool is fast, scalable, and capable of detecting intricate message-dependent deadlocks. Deadlocks in networks with thousands of components are detected within a few seconds.**

I. Introduction

In modern architectures, performance is gained by increasing parallelism [1]. Multi-Processor Systems-on-Chips (MPSoCs) integrate on a single die several processing, memory, and I/O devices. As bus performance degrades when the number of cores increases, complex Networks-on-Chips (NoCs) constitute an alternative solution for scalable interconnect infrastructures [2], [3]. Formal verification of NoCs is a challenge. In particular, deadlock freedom is a crucial property that also is difficult to automatically verify. A solution is to analyze abstract microarchitectural models of communication fabrics. A well-defined set of primitives – named xMAS for eXecutable MicroArchitectural Specifications – has been proposed by Intel to precisely describe these models [4]. Chatterjee and Kishinevsky developed techniques to generate inductive invariants and use these invariants to improve the performance of hardware model-checking of Verilog descriptions [5]. Recently, Gotmanov *et al.* proposed a Boolean encoding of deadlock equations [6]. Using these equations and automatically generated invariants, the authors were able to verify Verilog designs for deadlocks. Their techniques scale up to networks with hundreds of components and tens of queues. Actual designs typically consist of hundreds or even thousands of queues. We report results[1] on networks with thousands of components and hundreds of queues. A direct comparison with Intel's algorithms is not possible as their tools and benchmarks are not publicly available. We exhibit one example that is out-of-reach for Intel's techniques but is verified instantaneously by our algorithm.

Our novel deadlock detection algorithm is based on the following two key concepts. The wait-for relations of xMAS models are captured in a *labelled waiting-graph*. A deadlock is defined as a *feasible closed subgraph* of the waiting-graph. Our algorithm analyses each queue of a network and either stops if a blocking queue has been found or returns "no deadlock" when all queues have been visited. For each queue, a labelled waiting-graph is built. A deadlock is found when a feasible logically closed subgraph is found in the waiting-graph of a queue. Building the waiting-graph and searching for a feasible logically closed subgraph happen on-the-fly.

The next section briefly introduces the xMAS language and illustrates the difficulty of finding deadlocks in xMAS models. Section 3 presents the theoretical foundations of our algorithm which is detailed in Section 4. Section 5 demonstrates the applicability and the efficiency of our algorithm on several and distinct examples extracted from academic and industrial NoC designs. Both routing and message dependent deadlocks are detected within seconds in designs with thousands of components. Finally, Section 6 relates our work to Intel's approach and Section 7 concludes.

II. xMAS Models

We briefly introduce the xMAS language. Our presentation is inspired by the original xMAS paper where more details can be found [4].

An xMAS model is a network of primitives connected via typed data *channels*. A channel is connected to an *initiator* and a *target*. A channel is composed of three signals. Channel signal $x.irdy$ indicates whether the initiator is ready to write to channel x. Channel signal $x.trdy$ indicates whether the target is ready to read channel x. Channel signal $x.data$ contains data that are transferred from the initiator output to the target input if and only if both signals $x.irdy$ and $x.trdy$ are set to true. Figure 1 shows the eight primitives of the xMAS language. A *function* primitive manipulates data. Its parameter is a function that produces an outgoing packet from an incoming packet. Typically, functions are used to convert message types and represent message dependencies inside the

[1]The source code for the algorithm presented in the paper are available at http://www.cs.ru.nl/∼freekver/fmcad11/

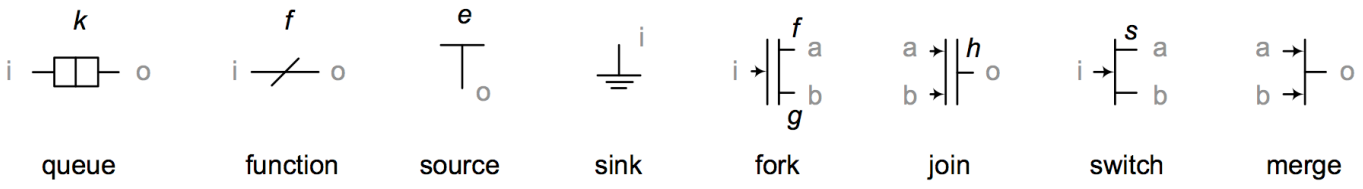


Fig. 1: Eight primitives of the xMAS language. Italicized letters indicate parameters. Gray letters indicate ports.

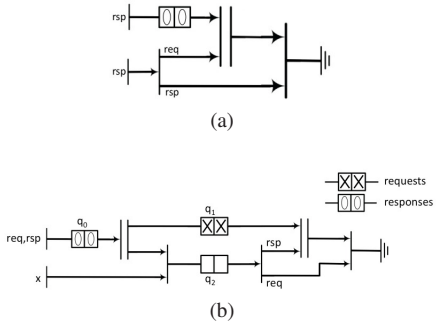


Fig. 2: Microarchitectural models

fabric or in the model of the environment. A *fork* duplicates an incoming packet to its two outputs. Such a transfer takes place if and only if the input is ready to send and the two outputs are both ready to read. As in previous publications [5], [6], we assume forks with identity functions. A *join* is the dual of a fork. The function parameter determines how the two incoming packets are merged. A transfer takes place if and only if the two inputs are ready to send and the output is ready to read. The function parameter must be total, i.e., a join is always able to produce a packet if both inputs are ready. A *switch* uses its function parameter to determine to which output an incoming packet must be routed. A *merge* is an arbiter. It grants its output to one of its inputs. A merge is fair, i.e., all inputs are served eventually. A *queue* stores data. As we assume fair arbiters, we abstract away from their internal state and a queue is the only state holding element. Messages are non-deterministically produced and consumed at *sources* and *sinks*. Sources and sinks are fair, i.e., messages are eventually created or consumed. A source or sink may process multiple message types. A *configuration* σ represents the current occupation of queues, i.e., the current state. The semantics of an xMAS network is specified using synchronous equations for each primitive [4]. Configurations are updated when messages are produced, consumed, or moved to a next queue. A *legal* configuration is a configuration where the buffer sizes of the queues are not exceeded. A configuration is *reachable* if it is possible to reach it starting from the empty network. A channel c *has type* p if and only if there exists a reachable configuration such that a packet p is located in channel c. The set of all types of channel c is noted $\tau(c)$.

Deadlocks are difficult to find in xMAS models as the traditional association between cycles and deadlocks is neither sufficient nor necessary. Consider the microarchitectural model in Figure 2b. One source emits both response and request packets. The type of packets of the other source is left uninterpreted for now. The first source feeds into queue q_0 which then enters a fork. The lower output of the fork is merged with the other source into queue q_2. From q_2, request packets are routed to a sink while response packets are joined with packets stored in q_1. The configuration in Figure 2b has a request packet in q_1 and a response packet in q_0. The join waits for response packets in q_2. Response packets wait for the fork. This fork waits for space in q_1 which in turn waits for the join. This completes a circular wait, but this circular wait is not necessarily a deadlock. If $x = \{rsp\}$, i.e., the second source generates response packets, the network is deadlock-free. If $x = \{req\}$, the configuration is a deadlock. Consider the microarchitectural model in Figure 2a. The queue waits for the join. The join waits for a request packet. As the source never produces a request packet, the configuration is a deadlock without circular waits.

III. Theoretical Foundations

Let Q be the set of queues in the network and let $q.out$ denote the output channel connected to queue q. A configuration is *stuck* if and only if the packets in all queues are blocked, i.e.:

$$\mathbf{stuck}(\sigma) \stackrel{\text{def}}{=} \forall q \in Q \cdot q.out.irdy \implies \neg q.out.trdy.$$

Definition 1: A configuration σ is a *deadlock configuration*, notation $\mathbf{dl}(\sigma)$, if and only if it is a non-empty configuration such that:

$$\mathbf{dl}(\sigma) \stackrel{\text{def}}{=} \mathbf{legal}(\sigma) \wedge \mathbf{reachable}(\sigma) \wedge \mathbf{stuck}(\sigma)$$

In a deadlock, the output channel of each queue that contains packets is blocked. None of the packets can proceed.

We formulate a set of *blocking equations*, notation $\mathbf{Block}(c, p)$, representing whether a packet p can be permanently blocked in channel c (Figure 4). We also define the *idle equations*, notation $\mathbf{Idle}(c, p)$, representing whether channel c can be permanently empty for packet p. We define a blocked queue, notation $\mathbf{BlockQ}(q)$, as a queue q containing a blocked packet.

$$\mathbf{BlockQ}(q) \equiv \exists p \in \tau(q.out) \cdot \#q.p \geq 1 \wedge \mathbf{Block}(q.out, p)$$

The equations in Figure 4 capture the reason why components are permanently blocking or idle. A queue is blocking if it is full and the component connected to its output channel is blocking ($\mathbf{full}(q)$ denotes "$\#q = q.size$"). A queue is idle for packet p either if it is empty and its input is connected to an idle component or if messages with packet p are blocked

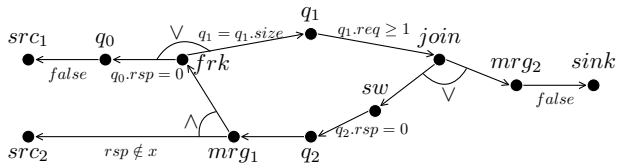


Fig. 3: Labelled waiting graph for the model in Figure 2b

Definition 2: Let c be a channel, let x (y) be the target (initiator) component of c, and let $x.in'$ ($x.out'$) be the other in- (out-) port of component x (y).

$$
\begin{array}{lcll}
\mathbf{Block}(c,p) & \equiv & \mathbf{full}(x) \wedge \mathbf{BlockQ}(x) & \text{iff } x = \text{queue} \\
& \equiv & \mathbf{Block}(x.out, f(p)) & \text{iff } x = \text{function} \\
& \equiv & \text{false} & \text{iff } x = \text{sink} \\
& \equiv & \mathbf{Block}(x.out_1, p) \vee \mathbf{Block}(x.out_2, p) & \text{iff } x = \text{fork} \\
& \equiv & \mathbf{Block}(x.out, p) \vee \forall p' \in \tau(x.in') \cdot \mathbf{Idle}(x.in', p') & \text{iff } x = \text{join} \\
& \equiv & \mathbf{Block}(x.out_1, p) & \text{iff } x = \text{switch} \wedge b(p) \\
& \equiv & \mathbf{Block}(x.out_2, p) & \text{iff } x = \text{switch} \wedge \neg b(p) \\
& \equiv & \mathbf{Block}(x.out, p) & \text{iff } x = \text{merge} \\
\mathbf{Idle}(c,p) & \equiv & \left\{ \begin{array}{l} y.p = 0 \wedge \mathbf{Idle}(y.in, p) \vee \\ \exists p' \in \tau(y.out) \cdot p \neq p' \wedge y.p' \geq 1 \wedge \mathbf{Block}(y.out, p') \end{array} \right. & \text{iff } y = \text{queue} \\
& \equiv & \forall p' \in \tau(y.in) \cdot f(p') = p \implies \mathbf{Idle}(y.in, p') & \text{iff } y = \text{function} \\
& \equiv & p \notin \tau(y) & \text{iff } y = \text{source} \\
& \equiv & \mathbf{Idle}(y.in, p) \vee \exists p' \in \tau(y.out') \cdot \mathbf{Block}(y.out', p') & \text{iff } y = \text{fork} \\
& \equiv & \mathbf{Idle}(y.in_1, p) \vee \mathbf{Idle}(y.in_2, p) & \text{iff } y = \text{join} \\
& \equiv & \mathbf{Idle}(y.in, p) & \text{iff } \left\{ \begin{array}{l} y = \text{switch} \wedge \\ (b(p) \iff c = y.out_1) \end{array} \right. \\
& \equiv & \mathbf{Idle}(y.in_1, p) \wedge \mathbf{Idle}(y.in_2, p) & \text{iff } y = \text{merge}
\end{array}
$$

Fig. 4: Blocking equations

by other packets and cannot leave the queue. Formally, the latter means that the channel written by the queue never receives packet p. A function is blocking if its output channel is blocking after application of the function. A function is idle for packet p if its input channel is idle for all packets for which the application results in p. A sink is never blocked. A source can be idle for a particular message type. A fork is blocked if one of its outputs is blocked. A fork is idle if its input is idle. A fork can also be blocked if an output channel is blocking, since a fork can only produce two packets if all its output channels are ready to receive. A join is blocked if its output is blocked or one of its inputs is idle for any packet. A join is idle if one of its inputs is idle. A switch has one blocking equation for each possible output. The first (second) output channel of a switch is idle for p if the condition (i.e., function s applied to packet p) does not (does) hold for p or its input is idle. A merge is blocked if its output is blocked. Note that a merge may also be blocking if the other input channel is selected. However, since we assume fair merges, this cannot permanently block the input channel. As our equations capture the reason why a component is *permanently* blocking an input channel, this blocking scenario need not be reflected in the deadlock equations of the merge. A merge is idle if *both* its inputs are idle.

We now prove[2] correctness of the deadlock equations, i.e., a configuration is stuck if and only if there is a blocked queue.

Lemma 1: There exists a non-empty stuck configuration if and only if for some queue q the blocking equations are feasible:

$$\exists q \in Q \cdot \mathbf{BlockQ}(q) \iff \exists \sigma \cdot \mathbf{stuck}(\sigma)$$

Configuration σ in Lemma 1 is a configuration in which all packets are blocked. The configuration is not necessarily legal or reachable. *Legality equations* (noted **Legal**) are added to bound the number of packets stored in queues. They have the following form: $\{"\#q \leq q.size" \mid q \text{ is a queue}\}$. To rule-out unreachable configurations, a *reachability invariant* (noted **Inv**) is automatically generated. We have made a quick re-implementation of the invariant generation technique used in [5]. In all examples presented in this paper, the invariants generated by our quick re-implementation were enough.

The next Lemma shows that if there is a deadlock then our algorithm will find it. Note that because we may output a deadlock that is not reachable, the other direction does not hold.

Lemma 2: For any set of invariants **Inv**, if there exists a deadlock configuration, then there exists a blocked queue q.

$$\exists \sigma \cdot \mathbf{dl}(\sigma) \implies \exists q \in Q \cdot \mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv}$$

[2]All proofs are available in an appendix at the end of this paper.

Given a queue q, the *labelled waiting graph* is a graph with as vertices the components of the network. Figure 3 shows this graph for queue q_1 in Figure 2b. The next Section details the efficient construction of this graph. Function $E_{\texttt{wait}}$ represents the edge function, i.e., $E_{\texttt{wait}}(x)$ returns the set of neighbors of component x. We let $\bigwedge(x)$ and $\bigvee(x)$ return true if and only if the edges going out of component x are conjunctive or disjunctive. An edge (x_0, x_1) between components x_0 and x_1 is labelled according to the deadlock equations of channel $x_0.out$ connecting these components. Starting from queue q the labels directly correspond to the set of equations $\mathbf{BlockQ}(q)$.

Definition 3: A waiting subgraph S is *logically closed*, notation $\mathbf{closed}(S)$, iff:

$$\mathbf{closed}(S) \stackrel{\text{def}}{=} \forall x \in S \cdot \begin{cases} \forall n \in E_{\texttt{wait}}(x) \cdot n \in S \text{ iff } \bigwedge(x) \\ \exists n \in E_{\texttt{wait}}(x) \cdot n \in S \text{ iff } \bigvee(x) \end{cases}$$

A subgraph S is *feasible* if and only if the conjunction of the constraints on all edges in S, the set of legality constraints, and the set of invariants, is feasible. For instance, subgraph $\{q_1, join, mrg_2, sink\}$ in Figure 3 is logically closed but not feasible. The next lemma shows that a deadlock is a feasible logically closed subgraph.

Lemma 3: For queue q, the deadlock equations are feasible if and only if the waiting graph of q contains a feasible and closed subgraph.

$$\forall q \in Q \cdot (\mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv} \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Our final theorem is a corollary from Lemmas 2 and 3.

Theorem 1: For any set of invariants $\mathbf{Inv}$, if there is a deadlock, then there exists a waiting subgraph that is feasible and closed.

IV. Algorithm

The algorithm detects closed subgraphs and determines their feasibility. It starts a search in some queue q_0 with some packet p. The current subgraph S under consideration is $\{q_0\}$. The search expands waiting neighbors, adding them to S, as long as the subgraph is open and feasible. The search starts with forward expansion. Each forward edge requires the next component to be permanently blocked. When encountering a join, the search proceeds both forwards to determine whether the output channel can be permanently blocked *and* backwards to determine whether the input channel can be permanently idle. The result is that a tree – spanning over the waiting graph – with as root q_0 is created on-the-fly. In case of a conjunctive component, unexplored edges are marked as 'open', since they must still be explored. The algorithm proceeds its search until S is closed. The algorithm keeps track of the set of equations $\mathcal{E}_{\texttt{curr}}$ of the path leading from the initial queue q_0 to the current component x.

If a cycle, a sink, or a source is encountered, the algorithm ends its recursion. If there are no open edges and if the current subgraph S is feasible, a deadlock has been found and the algorithm terminates. Otherwise, the algorithm backtracks to the latest disjunctive point. To prevent an exponential graph exploration, we implement a memoization technique. After each recursive call, the equations – named *closing equations* – of each path leading to a cycle or source are stored. If a component is encountered that has already been visited, a deadlock has been found if the conjunction of $\mathcal{E}_{\texttt{curr}}$ and the closing equations is feasible. This ensures that each component of the waiting graph has to be visited at most once.

Consider the network in Figure 2b. We let the algorithm start in queue q_1 with packet req. It will create the graph in Figure 3 on-the-fly. The algorithm starts with expanding the join, adding "$\#q_1.req \geq 1$" to $\mathcal{E}_{\texttt{curr}}$. There are two ways to proceed: forwards to mrg_2 or backwards to the switch. The algorithm proceeds forwards. As this leads to a sink, no deadlock is found. The algorithm associates the closing equation "false" to the sink. The algorithm then proceeds backwards to determine whether the switch can be permanently idle for packet rsp. Queue q_2 is expanded, adding "$\#q_2.rsp = 0''$ to $\mathcal{E}_{\texttt{curr}}$. The algorithm expands mrg_1. There are two ways to proceed: backwards to the fork or backwards to the source. The algorithm first expands the fork, but keeps track of the open edge to the source. Again, there are two ways to proceed: one forwards leading to queue q_1 and one backwards leading to queue q_0. The algorithm first proceeds forwards, adding "$\#q_1 = q_1.size$" to $\mathcal{E}_{\texttt{curr}}$. Queue q_1 has already been explored. Since there is one open edge, the algorithm starts propagating information upwards to the fork by associating "$\#q_1 = q_1.size$" as a closing equation for the fork. Consequently, it is removed from $\mathcal{E}_{\texttt{curr}}$. It proceeds by exploring queue q_0. Since this is connected to a source that injects rsp messages, queue q_0 cannot be idle for rsp. The algorithm associates closing equation "false" to the source and to queue q_0. This is propagated upwards. The closing equations of the fork become: "$\#q_1 = q_1.size \vee false$". The open edge from the merge to the source is explored. If we assume that src_2 injects rsp-messages, the algorithm associates closing equation "true" to src_2. This is propagated upwards, and the closing equation associated to the merge becomes $(\#q_1 = q_1.size \vee false) \wedge true$. As there are no more open edges, the algorithm checks the feasibility of the conjunction of $\mathcal{E}_{\texttt{curr}}$ and the closing equation of the merge, i.e., feasibility of $\{\#q_1.req \geq 1, \#q_2.rsp = 0, \#q_1 = q_1.size\}$. The solution to these equations corresponds to any deadlock configuration where q_1 is full with a request at its head, no responses are in q_2.

Algorithm 1 shows the pseudo code of our algorithm. This is one half of the algorithm, as function IdleDetect is needed to determine deadlocks for joins. Function IdleDetect is the exact dual of DeadDetect. The complete algorithm is a mutual recursion between these two dual functions. The algorithm takes four parameters: a component x that is to be explored, the current packet p, the number of open edges $open$ and the set of equations $\mathcal{E}_{\texttt{curr}}$. For each queue q, the closing equations are stored in $\mathcal{E}_{\texttt{closing}}[q]$.

Algorithm 1 DEADDETECT$(x, p, open, \mathcal{E}_{\texttt{curr}})$

1: **if** $x ==$ queue **then**
2: $\quad\mathcal{E}_{\texttt{curr}} \wedge\!\!= \text{“}\#x = x.size''$
3: $\quad$**if** $\mathcal{E}_{\texttt{closing}}[x] == \emptyset$ **then**
4: $\quad\quad\mathcal{E}_{\texttt{closing}}[x] = \text{“}false''$
5: $\quad\quad$**for all** $p' \in \tau(x.out)$ **do**
6: $\quad\quad\quad\mathcal{E}_{\texttt{curr}} \wedge\!\!= \text{“}\#x.p' \geq 1''$
7: $\quad\quad\quad$DEADDETECT$(x.out, p', open, \mathcal{E}_{\texttt{curr}})$
8: $\quad\quad\quad\mathcal{E}_{\texttt{curr}} \not\wedge\!\!= \text{“}\#x.p' \geq 1''$
9: $\quad\quad\quad\mathcal{E}_{\texttt{closing}}[x] \vee\!\!= \mathcal{E}_{\texttt{closing}}[x.out]$
10: $\quad\quad$**end for**
11: $\quad$**else if** $open == 0$ **then**
12: $\quad\quad$* Determine feasibility of equations *\
13: $\quad\quad$* Report deadlock if feasible *\
14: $\quad$**end if**
15: $\quad\mathcal{E}_{\texttt{curr}} \not\wedge\!\!= \text{“}\#x = x.size''$
16: **else if** $x ==$ function **then**
17: $\quad$DEADDETECT$(x.out, f(p), open, \mathcal{E}_{\texttt{curr}})$
18: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out]$
19: **else if** $x ==$ sink **then**
20: $\quad\mathcal{E}_{\texttt{closing}}[x] = \text{“}false''$
21: **else if** $x ==$ fork **then**
22: $\quad$DEADDETECT$(x.out_1, p, open, \mathcal{E}_{\texttt{curr}})$
23: $\quad$DEADDETECT$(x.out_2, p, open, \mathcal{E}_{\texttt{curr}})$
24: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out_1] \vee \mathcal{E}_{\texttt{closing}}[x.out_2]$
25: **else if** $x ==$ join **then**
26: $\quad$DEADDETECT$(x.out, p, open, \mathcal{E}_{\texttt{curr}})$
27: $\quad$**for all** $p' \in \tau(x.in')$ **do**
28: $\quad\quad$IDLEDETECT$(x.in', p', open, \mathcal{E}_{\texttt{curr}})$
29: $\quad$**end for**
30: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out] \vee \mathcal{E}_{\texttt{closing}}[x.in']$
31: **else if** $x ==$ switch **then**
32: $\quad$**if** cond(p) **then**
33: $\quad\quad$DEADDETECT$(x.out_1, p, open, \mathcal{E}_{\texttt{curr}})$
34: $\quad\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out_1]$
35: $\quad$**else**
36: $\quad\quad$DEADDETECT$(x.out_2, p, open, \mathcal{E}_{\texttt{curr}})$
37: $\quad\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out_2]$
38: $\quad$**end if**
39: **else if** $x ==$ merge **then**
40: $\quad$DEADDETECT$(x.out, p, open, \mathcal{E}_{\texttt{curr}})$
41: $\quad\mathcal{E}_{\texttt{closing}}[x] = \mathcal{E}_{\texttt{closing}}[x.out]$
42: **end if**

We detail the case where x is a queue. Other cases are processed similarly. In the case of a queue, x must be full in order to be blocking. Equation "$\#x = x.size$" is conjunctively added to the current set of equations (line 2). For each new packet p', the algorithm adds equation "$\#x.p' \geq 1$" and recursively determines whether the next component can be permanently blocking (lines 5–7). After the recursive call, equation "$\#x.p' \geq 1$" is retracted (line 8). After all recursive calls, equation "$\#x = x.size$" is retracted (line 15).

The number of open edges can increase only in IDLEDETECT. Open edges occur with functions, switches, and merges. Only if the number of open edges is equal to zero and if some cycle has occurred, the sets of equations are fed to a linear programming solver. We use lp_solve, an off-the-shelve linear programming solver [7]. We have equations stored in efficient data structures in such a way that, e.g., $(\#q_1 = q_1.size \vee false) \wedge true$ is stored simply as $\#q_1 = q_1.size$. Adding equations to this data structure is only possible if the resulting set of equations is still internally feasible, i.e., feasible without further invariants. This prevents unnecessary exploration of infeasible paths.

Correctness of the algorithm means that function DEADDETECT returns true if and only if there is a feasible closed subgraph.

Lemma 4:

$$\exists x, p \cdot \text{DEADDETECT}(x, p, 0, \emptyset) \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Remarks:

Counterexamples: If our algorithm finds a feasible and closed subgraph, it has given the set of constraints corresponding to this subgraph to a linear programming solver. This solver not only returns a boolean value indicating that the set of constraints is feasible, but also a solution. This solution assign integers to queues and headers. It is a detailed representation of a counterexample, i.e., a deadlock configuration.

Running time: Each separate run of the algorithm visits each component at most once. As per component deadlock equations are memoized there is no need to re-explore a visited component. The algorithm is executed once for each queue. The number of recursive calls is therefore $O(Q \cdot C)$ with Q the number of queues and C the number of components.

Before running the algorithm, the typing information needs to be computed, i.e., we need to compute $\tau(c)$ for all channels c. To obtain this information we perform exhaustive simulations. For each source and for each possible packet p injected at the source, we simulate the injection in an empty network until it is consumed. During this simulation p is added to $\tau(c)$ for each visited channel c. During this simulation, queues may need to be visited more than once.

Consider the network in Figure 6. The network is deadlock-free. To establish this, it must be established that always eventually a packet "5" arrives at queue q_1. During the simulation of packet "0" in source src_0, queue q_0 is visited 6 times. This establishes that $\tau(c) = \{0, 1, \ldots, 5\}$. Using this information, our algorithm needs to visit queue q_0 just once to establish deadlock freedom of the network.

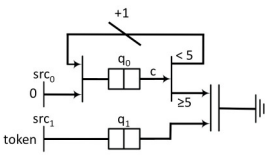


Fig. 6: xMAS model

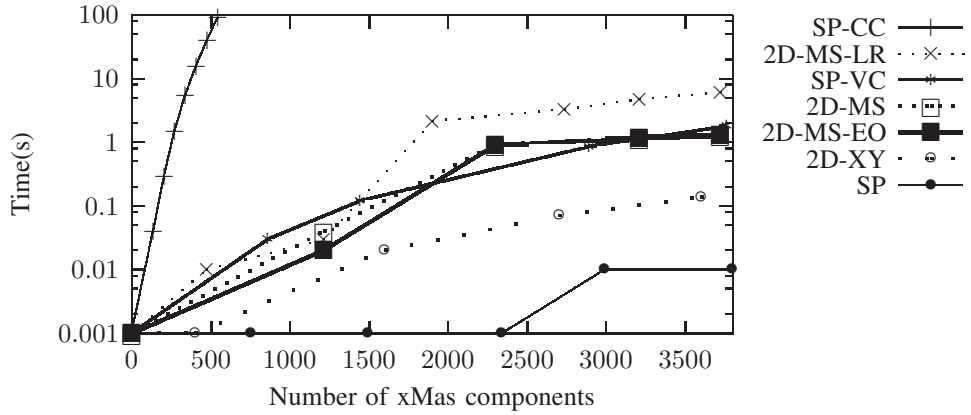


Fig. 5: Experimental results

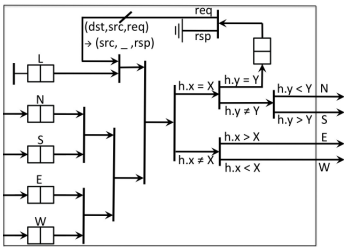
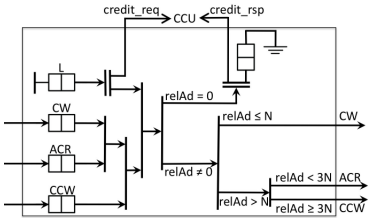
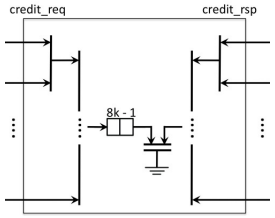


Fig. 7: xMAS model of an HERMES node



(a) Ring node with shortest path routing



(b) Credit control unit

Fig. 8: Spidergon with flow control

V. Experimental results

We consider two Network-on-Chip (NoCs): The HERMES NoC [8] from the University of Rio Grande in Brazil and the Spidergon NoC from STMicroelectronics [9]. All experiments have been performed on a Ubuntu machine with a 2.93 GHz Intel Core 2 Duo processor and 2 GB memory. Figure 5 gives an overview of experimental results on the benchmarks described hereafter.

HERMES is a two-dimensional mesh using XY routing [10]. Figure 7 shows an xMAS specification of a processing node with coordinates (X, Y). This node is a "slave". It introduces message dependencies as responses are generated upon reception of requests. A master node would only inject requests in its local queue and consume responses. We experimented with different layouts of masters and slaves: no master and no slave (curve 2D-XY), all nodes are both master and slave (curve 2D-MS), masters on the left part of the mesh and slaves on the right part (curve 2D-MS-LR), or masters on even columns and slaves on odd columns (curve 2D-MS-EO). Two layouts only are deadlock-free (2D-XY and 2D-MS-LR). The results show good performance for detecting deadlocks and proving their absence.

Spidergon is a ring where each processing node can send messages clockwise, counter clockwise, or across. Shortest path routing is used. At each node, the routing decision is based on the relative address $relAd = (d - s) \mod N$. Here d is the destination, s is the current node, and N is the total number of nodes. Because of the ring, this architecture has a deadlock (curve SP). In this case, performance is linear in the size of the ring.

To resolve this deadlock, virtual channels [11] are inserted to the right upper quarter of the ring only (curve SP-VC). The routing function is modified such that virtual channels are only used for each destination inside the quarter, other messages still use the regular channels. This case is slightly more difficult because there is no deadlock. If virtual channels are wrongly designed, deadlock detection is as in curve SP.

Another approach is to add a credit control unit (CCU, Figure 8) limiting the number of packets in the ring to $N \cdot k - 1$, where N is the size of the ring and k the size of the queues. When injecting messages in local queues, these messages are duplicated and sent to the CCU. When messages are sunk, they are also duplicated and sent to the CCU to free space. This unit may look unrealistic but its main purpose is to illustrate a difficult case for our algorithm. Indeed, the merges force our algorithm to branch on all the inputs of these merges. As it can be seen in curve SP-CC, this case is much harder. Still, networks with tens of agents and hundreds of components can be proven deadlock-free within a few minutes. If the counter

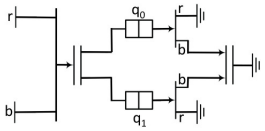


Fig. 9: Example from Intel

is wrongly sized, e.g., `cc_queue.size` $= Nk$, a deadlock is found as in curve SP.

Figure 9 shows a network abstracted from a real design from Intel [12]. The sources in the network emit red and blue tokens respectively. These tokens are duplicated into two queues. Red tokens are sunk, blue tokens are joined and then sunk. The network is deadlock-free, as queues q_0 and q_1 are fed with tokens in the same order. Given invariants automatically generated by [5], the approach of [6] cannot handle this example while our algorithm returns "no deadlock" instantaneously.

VI. Related work

We define a deadlock configuration while Gotmanov *et al.* [6] define a dead channel, i.e., a channel that is never idle but always blocked in some execution. Assuming fair merges and that a dead channel coincides with a blocked queue, the two definitions are logically equivalent. We can prove that there exists a dead channel if and only if there exists a deadlock configuration. Our approach covers a similar property as [6]. An important difference is that we directly tackle xMAS models and not their Verilog implementation. The two techniques are complementary. Our tool can be used to quickly remove all deadlocks in xMAS models before proving the Verilog deadlock-free.

VII. Conclusion

We have shown that based on the notions of a labelled waiting graph and a logically closed subgraph it is possible to efficiently detect deadlocks in microarchitectural models of communication fabrics. We demonstrated the applicability and efficiency of our solution on several deadlock avoidance mechanisms used in academic and industrial NoCs designs. Deadlocks are found within seconds in networks with thousands of components. We exhibited an example that can be proven deadlock-free using our technique but could not be handled by Intel's recent related solution. Our technique uses less and simpler invariants showing that using the labelled waiting graph we capture more information about the structure of xMAS models.

Acknowledgments

This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811. This research received a gift from Intel Corporation.

References

[1] W. Dally, "The end of denial architecture," Keynote at Design Automation Conference (DAC'09), 2009.
[2] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Proceedings of the Design Automation Conference*, Las Vegas, NV, 2001, pp. 684–689.
[3] L. Benini and G. D. Micheli, "Networks on Chips: A New SoC Paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.
[4] S. Chatterjee, M. Kishinevsky, and U. Ogras, "Quick formal modeling of communication fabrics to enable verification," in *Proc. of High Level Design Validation and Test Workshop (HLDVT'10)*, 2010, pp. 42–49.
[5] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," in *Proc. of Computer Aided Verification (CAV'10)*, 2010, pp. 321–338.
[6] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying deadlock-freedom of communication fabrics," 2011, to appear in Proceedings of VMCAI '11.
[7] M. Berkelaar, K. Eiklan, and P. Notebaert, *lp_solve (version 5.5.2.0)*, available under GNU LGPL. [Online]. Available: http://lpsolve.sourceforge.net/5.5/
[8] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 69 – 93, 2004.
[9] F. K. A. Karim and S. Dey, "An interconnect architecture for networking systems on chips," *IEEE Micro*, pp. 36–45, 2002.
[10] L. Ni and P. Mckinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, vol. 26, pp. 62–76, Februari 1993.
[11] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi, *Design of Cost-Efficient Interconnect Processing Units: Spidergon STNoC*, 1st ed. CRC Press, Inc., 2008.
[12] S. Chatterjee and M. Kishinevsky, personal communication.

Appendix
Proofs of Lemmas

Lemmas in Section 3

Lemma 1: There exists a non-empty stuck configuration if and only if for some queue q the blocking equations are feasible:

$$\exists q \in Q \cdot \mathbf{BlockQ}(q) \iff \exists \sigma \cdot \mathbf{stuck}(\sigma)$$

Proof:
($\Longleftarrow$) Assume queue q is non-empty and blocked in configuration σ. We prove that σ satisfies the set of blocking equations by structural induction on the definition of $\mathbf{Block}$. Thus σ is a solution of these equations, implying they are feasible.

The base case is trivial. For the inductive case, we proceed by case distinction and only detail the case where x is a queue. The other cases are similar. Channel $x.in$ is permanently blocked by assumption. This happens only when x is full. Thus σ satisfies the equation $\#x = x.size$. Furthermore, the packet at the head of x must be permanently blocked. Let p' denote the header of this packet. Channel $x.out$ must be permanently blocked for packet p', since otherwise the packet at the head of x eventually is removed from the queue and channel $x.in$ becomes alive, contradicting the assumption that $x.in$ is permanently blocked. By induction hypothesis, if channel $x.out$ is permanently blocked for p', σ satisfies $\mathbf{Block}(x.out, p')$.
($\Longrightarrow$) Assume that for some queue q the equations are feasible. This means there exists a solution, which is an assignment of integers to queues and packets. This solution is thus a configuration. We prove that in this configuration, each

channel c involved in the set of equations is permanently blocked by its target component x. The proof is again by induction on $\mathbf{Block}$ and then by case distinction. We provide details on the case of the join. By the induction hypothesis we know that either the output is permanently blocking join x, or the other input channel is permanently idle. In both cases, channel c is permanently blocked by the join. This concludes the proof that the target of the channel is permanently blocked. ■

Lemma 2: For any set of invariants $\mathbf{Inv}$, if there exists a deadlock configuration, then there exists a blocked queue q.

$$\exists \sigma \cdot \mathbf{dl}(\sigma) \implies \exists q \in Q \cdot \mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv}$$

Proof: From Lemma 1, we know that a configuration that is stuck implies a blocked queue. By definition, a legal configuration implies the legality constraints. A reachable configuration implies the reachability invariant. ■

Lemma 3: For queue q, the deadlock equations are feasible if and only if the waiting graph of q contains a feasible and closed subgraph.

$$\forall q \in Q \cdot (\mathbf{BlockQ}(q) \wedge \mathbf{Legal} \wedge \mathbf{Inv} \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Proof:
($\Longrightarrow$) By assumption, the set of blocking equations is feasible for queue q and packet p. Consider the set of equations $\mathcal{E}$ obtained by replacing each disjunction in $\mathbf{Block}(q.out, p)$ by one feasible operand of the disjunction. Let S be the waiting graph corresponding to $\mathcal{E}$. S is a subgraph from the waiting graph of q. Since $\mathbf{Block}(q.out, p)$ is feasible, S is feasible as well. Finally, by construction S contains all its conjunctive neighbors and exactly one disjunctive neighbor for each disjunctive component. Thus S is a feasible and closed waiting subgraph.
($\Longleftarrow$) Assume a feasible and closed subgraph S in the waiting graph of queue q. Let $\mathcal{E}$ be the set of equations corresponding to S. The set of equations $\mathcal{E}$ is a subset of $\mathbf{Block}(q.out, p)$ for some p. We prove that the feasibility of $\mathcal{E}$ implies feasibility of $\mathbf{Block}(q.out, p)$. Adding disjunctive operands to a disjunction somewhere in $\mathcal{E}$ can make it infeasible only if the number of operands is equal to zero. This is not possible since – as S is closed – $\mathcal{E}$ contains at least one disjunctive neighbor for each component. Adding conjunctive operands can make a conjunction infeasible, but since S is closed it already contains all its conjunctive neighbors. The feasibility of $\mathcal{E}$ implies the feasibility of the deadlock equations of q. ■

Lemmas in Section 4

Lemma 4:

$$\exists x, p \cdot \textsc{DeadDetect}(x, p, 0, \emptyset) \iff \exists S \cdot \mathbf{feasible}(S) \wedge \mathbf{closed}(S))$$

Proof: The algorithm reports a deadlock only at line 13. It keeps at all time track of the number of open edges in parameter *open*. As line 13 is only reached when the current subgraph is closed, i.e., $open == 0$, and when the linear program solver has determined feasibility, partial correctness is trivial to prove. What remains to be proven is termination. As the algorithm keeps track of visited components, each component is visited at most twice (once in DEADDETECT and once in IDLEDETECT). Thus the algorithm terminates. ■

Relation to Intel's approach

We have the following assumptions: 1) The network is livelock free, 2) the network is starvation free, and 3) a blocked channel implies a blocked packet in some queue.

We first prove a Lemma on *draining* a configuration. Let σ be a configuration. Draining σ is defined as:

- Canceling all further injections at the sources;
- Having the sinks consume all packets deterministically;
- Let the network execute until no packet in the network can be moved, i.e., until $\neg c.trdy$ for all channels c.

Lemma 5: Let σ be a legal and reachable configuration. Draining σ yields a unique legal and reachable configuration, denoted with $\mathbf{drain}(\sigma)$.

$$\mathbf{legal}(\sigma) \wedge \mathbf{reachable}(\sigma) \implies \mathbf{legal}(\mathbf{drain}(\sigma)) \wedge \mathbf{reachable}(\mathbf{drain}(\sigma))$$

Proof: Any configuration obtained from an execution starting in a legal and reachable configuration is legal and reachable. Non-determinism occurs at sources and sinks only. Since sources do not inject any further packets, and since sinks are deterministic while draining, no non-determinism occurs. Draining is a deterministic process and thus it suffices to show termination to show that it yields a unique configuration. By Assumption 1 no packet can move around infinitely in the network. Eventually, all packets will either be permanently blocked or arrive at a sink. Thus draining terminates. ■

Lemma 6: There exists a dead channel if and only if there exists a deadlock configuration:

$$\exists c \cdot \mathbf{dead}(c) \iff \exists \sigma \cdot \mathbf{dl}(\sigma)$$

Proof:
($\Longrightarrow$) Let c be a dead channel in some execution S. We know that $S \models \Diamond(c.irdy \wedge \Box\neg c.trdy)$. From the semantics of $\Diamond$, we can split S in to execution S_1 and S_2 such that $S = S_1S_2$, S_1 is a finite execution, and $S_2 \models c.irdy \wedge \Box\neg c.trdy$. Let σ' be the configuration obtained after execution of S_1. Let $\sigma = \mathbf{drain}(\sigma')$. In other words, replace execution S_2 by draining. By Lemma 5, σ is legal and reachable. By definition of draining, σ is stuck: either there are no more packets in the network in which case the network is stuck trivially, or all packets are blocked. What remains to be proven is that σ is non-empty. In execution S_2, channel c is permanently blocked. Execution S_2 is replaced by draining. This preserves the permanent blocking of channel c. Channel c can either be permanently blocked by a starvation scenario or because of a local deadlock. By Assumption 2, only the second can occur. This local deadlock is not resolved by draining, as the packets participating in this local deadlock are permanently blocked. Therefore, there is at least one channel that is blocked in σ. By Assumption 3 there is at least one queue blocked, meaning

that σ is non-empty. As σ is non-empty, legal, reachable, and stuck, σ is a deadlock configuration.
($\Longleftarrow$) As there exists a deadlock configuration, there exists a non-empty queue q which is permanently blocked. Channel $q.out$ is dead: as queue q is non-empty, the initiator of $q.out$ is not idle. As the packet in queue q cannot move, the target of $q.out$ is permanently blocked. ■