Modeling a Cache Coherence Protocol with the Guarded Action Language

Quentin Meunier, Yann Thierry-Mieg, Emmanuelle Encrenaz
Laboratoire d’Informatique de Paris 6, Sorbonne Université, Paris.
The TeraScale Architecture TSAR

- Hardware architecture designed to scale to up to 1024 core
- Hardware enabled cache coherence, logically a single address space, NUCA characteristics
Architecture

- Asynchronous process communicating over unidirectional shared channels
- Separate channels for direct and coherence transactions
Accessing memory

- Five independent networks in V5, six in V4

<table>
<thead>
<tr>
<th>Channel</th>
<th>Source</th>
<th>Dest.</th>
<th>Messages</th>
<th>Adr</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL1DTREQ</td>
<td>Proc</td>
<td>L1</td>
<td>DT_RD</td>
<td>1</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DT_WR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1PDTACK</td>
<td>L1</td>
<td>Proc</td>
<td>ACK_DT_RD, ACK_DT_WR</td>
<td>1</td>
<td>/</td>
</tr>
<tr>
<td>L1MCDTREQ</td>
<td>L1</td>
<td>L2</td>
<td>RD, WR</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L1MCCUREQ</td>
<td>L1</td>
<td>MC</td>
<td>CLNUP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MCL1CUACK</td>
<td>MC</td>
<td>L1</td>
<td>ACK_CLNUP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1MCCPACK</td>
<td>L1</td>
<td>MC</td>
<td>ACK_M_UP, ACK_B_INV, ACK_M_INV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCMEMDTREQ</td>
<td>MC</td>
<td>MEM</td>
<td>PUT, GET</td>
<td>1</td>
<td>/</td>
</tr>
<tr>
<td>MEMMCDTACK</td>
<td>MEM</td>
<td>MC</td>
<td>ACK_PUT, ACK_GET</td>
<td>1</td>
<td>/</td>
</tr>
</tbody>
</table>
Distributed Hybrid Cache Coherence Protocol DHCCP

- L2 cache maintains a directory of L1 copies of the data
  - Directory is physically distributed
  - Inclusive: any data in a L1 is necessarily in L2
  - Write through: L2 version is always the latest

- Direct transactions
  - Read, Write, Load-Linked/Store Conditional LL/SC, Compare and Swap CAS

- Coherence transactions
  - Update or evince L2 => update/invalidate all copies, wait for ACK
  - Multicast update if few copies
  - Broadcast an invalidate request if above the DHCCP threshold
  - Count the responses in both cases

- Hybrid Multicast/Broadcast policy based on DHCCP threshold
Design issues

- Separate Networks, Asynchronous behaviors...
- Errors are easy to make, hard to detect by simulation and testing
- This V4 example deadlocks...
Applying model-checking

- Could formal verification help gain more confidence in the design?

Challenges:
- Abstract from the real system faithfully
- Wide configuration space:
  - Number of cores/threads, Number of addresses, DHCCP threshold
  - Several versions of the protocol (V4 and V5)
- Smallest complete behavior: 3 cores, 2 addresses, threshold = 2
  - Observe both broadcast and multicast

Goal is automatic verification => model-checking
- Counter-example traces help debug
Verifying the protocol

- Extract manually from the code + specifications
  - Communicating automata over channels
  - Components: Processor, L1 cache, L2 cache, Memory
Building a model with Promela/SPIN

- Two Master 1 students: M. Najem 2011, A. Mansour 2012
- Build the Promela model
  - Formalisms of Communicating process matches the need

```plaintext
:: L1MCCUREQ ? m.type, eval(line_addr), m.cache_id ->
    do // Delete the cache id that did the request from the list of copies
       :: (cpt == CACHE_TH) -> break ;
       :: ((cpt < CACHE_TH) && (v_c_id[cpt] == VALID) && (c_id[cpt] == m.cache_id)) ->
            v_c_id[cpt] = INVALID;
            n_copies = n_copies - 1;
            break;
       :: else -> cpt = cpt + 1;
    od;
```
Results with SPIN

- Initial models are too detailed
  - Observation automata are encoded into the model to check its properties
  - Cumbersome/intrusive observation mechanism for channels
  - Incremental modeling of each component + verification in isolation is possible
  - Parametric features are good
  - Simulator and traces as sequence diagrams are very useful
- Two versions of the protocol modeled
  - More aggressive data abstraction in the second version
  - Some extensions explored e.g. LL/SC
- Full verification only possible for very small configurations
  - Unable to obtain full formal verification
  - POR reductions limited by heavy channel usage
Modeling and Verification in DiViNe

- Master 2 student: Z. Gharbi
- DiViNe is both a language and a model checker
  - Several versions, now focused on code verification
  - BEEM benchmark (2007) -> LTSmin, ITS-tools, Divine…
- Similar in concept, but much more basic than Promela
  - Parametric constructions with m4 preprocessor
  - Channel support proved inadequate: use global variables
- Properties encoded as LTL with fairness
  - Only Divine itself supports the keyword !
- Able to reproduce the deadlock + patch
  - Still unable to model-check truly relevant configurations
  - Integration of other tools a bit limited
Modeling in Guarded Action Language

- Master 2 student: D. Zhao
- GAL is an intermediate pivot language for concurrent semantics
  - Integers, and fixed size arrays of integers
  - Parametric and compositional features
- Initially supported by a powerful SDD engine (lots of MCC medals)
  - Additional support now for LTSMin+POR
  - Some SMT based verification
A simple GAL

gal simple {
    int a = 5 ;
    int b = -2 ;
    array [3] tab = (0, 8, -6);

    transition t1 [ a < tab [2] ] {
        a = (b + 3) * 255;
        b = a * tab [1];
        self."act";
        self."act";
    }

    transition t2 [true] label "act" {
        tab [0] = (tab [0] - 1) | ((tab [0] == 255) * 255);
    }

    transition t3 [true] label "act" {
    }
}

property goal [reachable] : tab[0] == 8;
Composite and Parametric features

- Instantiation of components
- Parameters over finite range
  - For loop
  - Parametric transitions and labels

```gal
int a = 0;
transition t1 [a < 5] label "label_t1" {
    a = a + 1;
}

composite compo {
    simple spl1;
    simple spl2;
    synchronization s1 label "label_s1" {
        spl1."label_t1";
        spl2."label_t1";
    }
}
```
Modeling with GAL

- Explicit models of channels
  - Two variants depending on data
- Automata directly expressed with a « state » variable
  - Labels used to describe channel operations
- Description is hierarchical and parametric
  - Composite description makes use of arrays of cores+L1; arrays of L2 …
- Fine control over atomicity semantics
  - Fusion of REQ/ACK in some scenarios
- No simulator
  - « Unit » verification used to debug model behavior
« Unit verifying »
Verification with ITS-Tools

- Performance sensitive to the description
  - Decomposition/recomposition heuristics still WIP
- With appropriate descriptions and hierarchy, full verification is possible
  - First full result on the minimal target configuration 3/2/2
  - Scale up is still limited, largest configurations 3/3/3, 4/2/2, 6/1/2… even with 24h and sizeable RAM
- No deadlocks reported in any configuration
- Full LTL with fairness results still incomplete
- Data abstraction prevents verification of memory model consistency in this version
Conclusion

- Formal modeling/verification is still a costly proposition
  - Manual abstraction is not very trustworthy, but…
  - Modeling all the implementation details swamps the model
  - Protocol issues are not necessarily in the routing/transport details
- Different solution engines/tools have different strengths and weaknesses
  - Lack of a more uniform description language, well supported by several tools (e.g. SMT equivalent)
- Model-checking was part of the result
  - A lot of confidence and understanding was also gained purely by building the formal descriptions themselves and debugging them