MPSoC '17

Accurate Study and Optimization of Synchronization Barriers in a NoC based MPSoC Architecture

Frédéric Rousseau

TIMA lab – University of Grenoble Alpes

A joint work with Maxime France-Pillois et Jérôme Martin
CEA LETI

Where are the synchronization barrier ?

```
#include <omp.h>
#define TAB_SIZE 1000
int main (void)
    {
    unsigned int n=0;
    unsigned int sinTable[TAB_SIZE];

    omp_set_num_threads(16);
    #pragma omp parallel for shared (sinTable)
    for (n = 0; n < TAB_SIZE; n++)
        sinTable[n] = n * 2;


    print_table(sinTable);
    return 1;
    }
```

Parallelization of 1000 computations on 16 threads

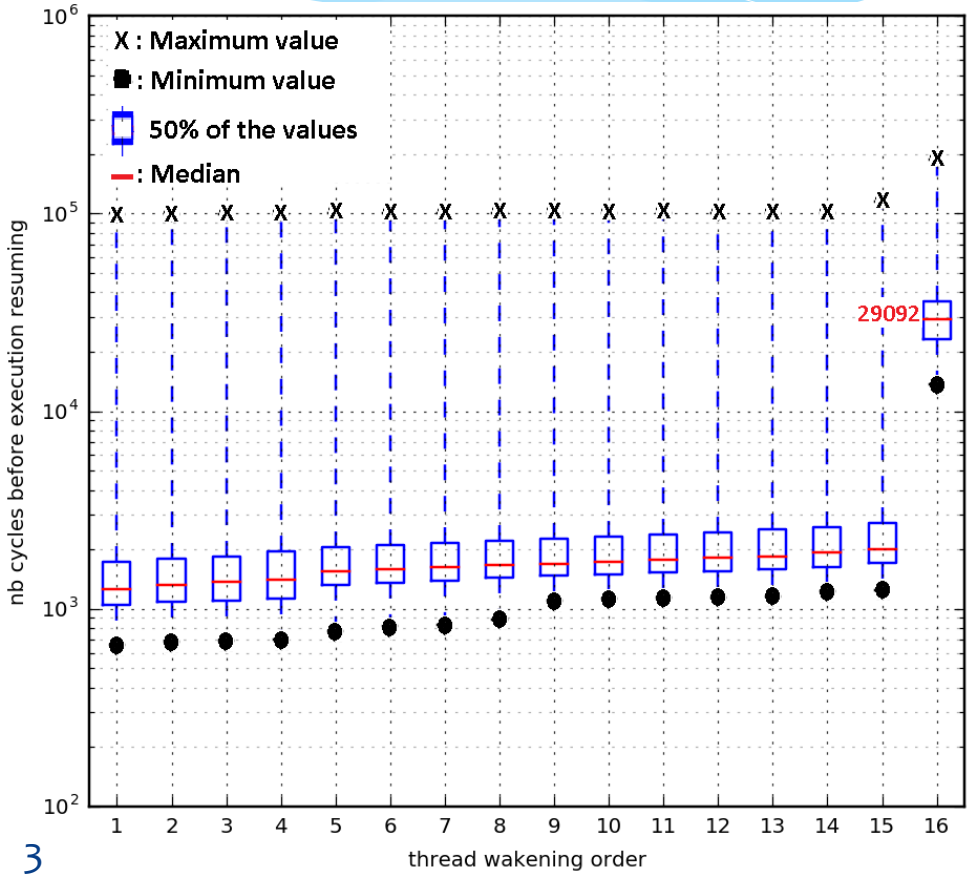
Implicit synchronization barrier

Print results

But it introduces a delay in the execution

Motivation through an example

* The figure represents the release delay by thread
* The Y-axis represents the number of cycles between the arrival of the last thread to the barrier, and the time a thread leaves the barrier to resume its nominal execution.
* The last thread takes 29092 cycles to resume its execution



Agenda

1. Definitions, motivations, and challenges
2. Efficient emulation environment
3. Observations and optimizations for active wait
4. Observations and optimizations for passive wait
5. Conclusion and future work

More details of synchronization barriers

* 2 kinds of delays are introduced by synchronization barriers
  * Application dependant delays (long time to completion for one task)
  * ***Intrinsic delays*** of the synchronization itself

* In a barrier, a thread waits until the others get ready
  * ***Active wait*** (or busy wait): polling on a waiting flag
  * ***Passive wait***: the waiting thread is put in sleeping mode
    * After a predefined amount of time for GNU OpenMP
    * Usually based on ***Futex*** in Linux ("Fast Usermode muTEX"), it may request relatively expensive system calls to manage operations on the wait queue.

Challenges for optimizations

* How to get an accurate measurement of the time spent in synchronization barriers ?

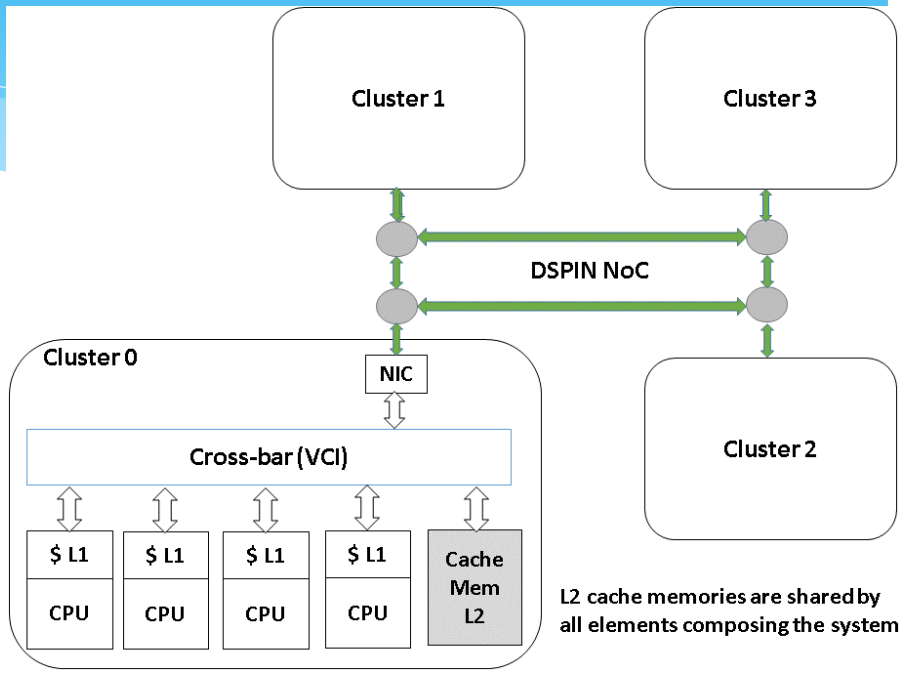
  * Usually, it consists in code instrumentation to extract timing information … and it affects the program behavior itself …

  * So our idea is to provide a solution:
    * ***Efficient*** (for observation and measurement)
    * ***Effortless*** for software (application) developpers

  ➔ We have developped a ***non-intrusive measurement tool chain***

Experimentation Environment

* Full coherent shared memory manycore platform
  * MIPS32 processor (private L1)
  * L2 cache is a shared memory
  * Cross-bar VCI protocol in cluster
  * DSPIN NoC between clusters



* Evaluation platform
  * Veloce2 Quattro emulator
    * Full RTL system with cycle accurate precision
    * Port and boot of Linux 4.6 (and µClibC)
    * Use of gcc for app. and OpenMP library compilation
  * 8, 16 and 24 core architectures have been emulated

A non-intrusive measurement toolchain

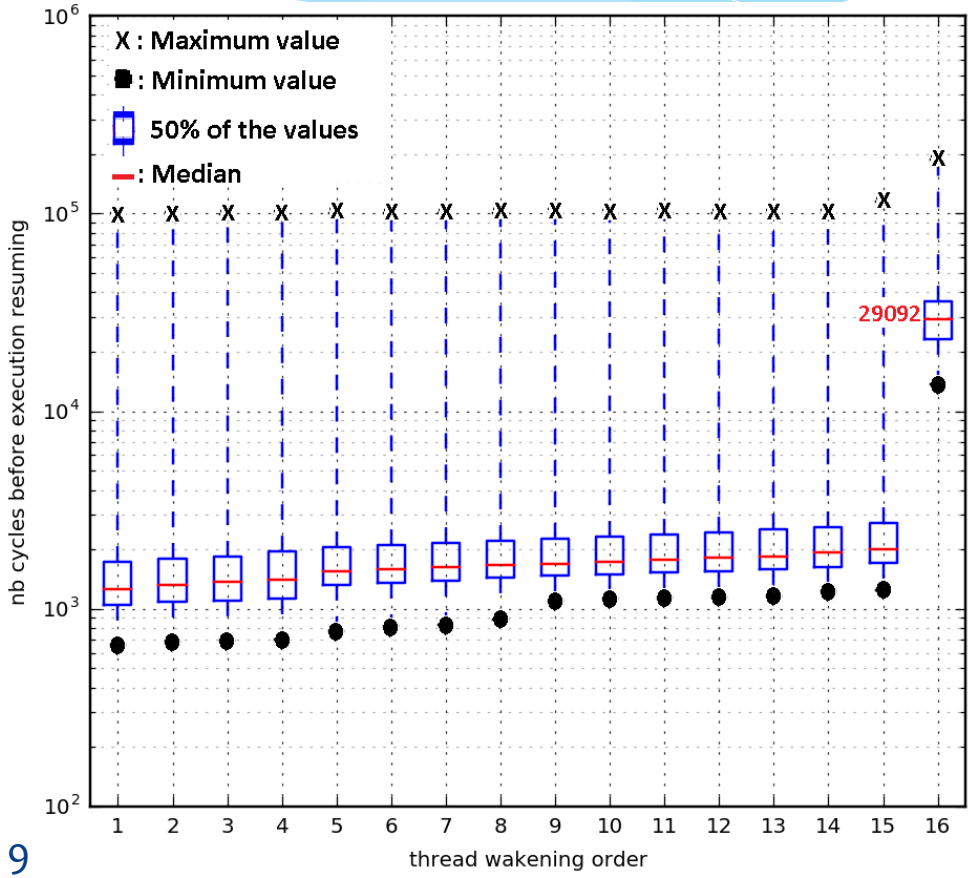
* Thank to the communication between the emulator and a workstation:
  * Extraction of useful signals (CPU registers, …) at runtime
    * These signals are dumped into files to be analyzed later
  * Such monitors do not disturb the nominal execution flow of the program
* ***No modification of the original application source code***

* Off course, it requires:
  * some modifications of the RTL platform to implement monitors
  * SW tools to follow function calls and make timing analysis

Observations for active wait

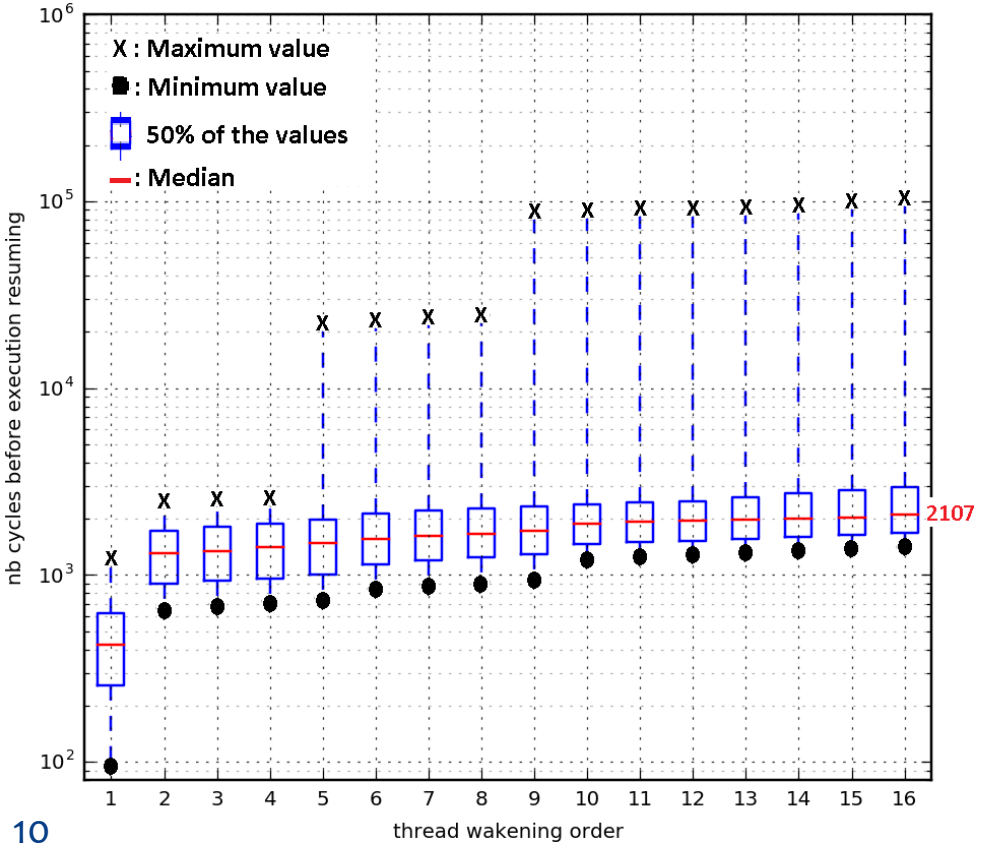
* The last thread takes 29092 cycles to resume its execution

* Thank to our measurement tool chain, we identified a ***contention issue*** in the DSPIN network (L2 memory is not able to serve all requests)
  * Kernel accesses
  * Periodic polling of the waiting flag
  * Managing the Futex list
    * The function call stack reveals that about 28000 cycles are spent ***on managing Futex (passive wait)***



Optimizations for active wait

* Modification of the GNU OpenMP library removing the ***Futex management (passive wait)*** when threads are waiting only in active wait mode
  * It reduces the contention troubles in L2 memory
* The last arriving thread is the first to resume



Optimizations for active wait

* Huge gain for this optimization

Threads number	full release phase delay without optimization	full release phase delay with optimization	Gain
8 on 8 cores	5425 cycles (median)	656 cycles (median)	88%
16 on 16 cores	29092 cycles (median)	2107 cycles (median)	93%
24 on 24 cores	176738 cycles (median)	7444 cycles (median)	96%

* This optimization is transparent for the SW developer
* Such optimizations seem to be implemented on LLVM OpenMP (not available for our experiment environment).

Observations for passive wait

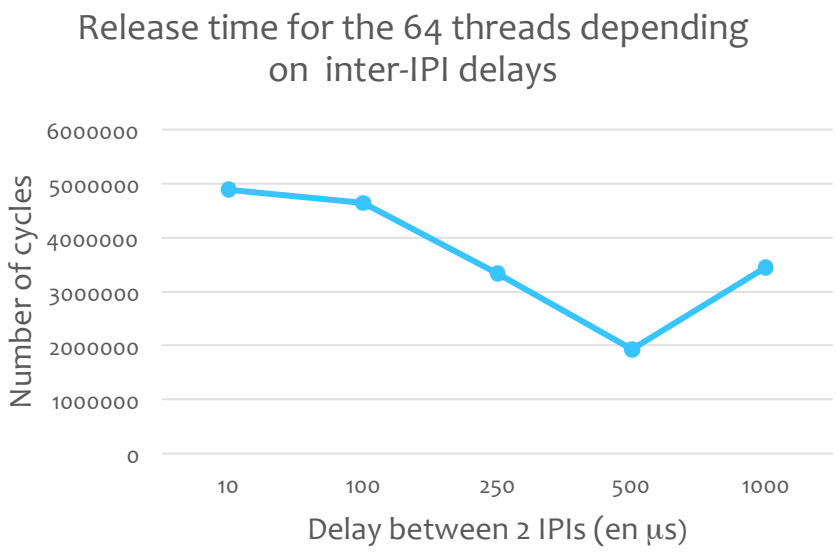
* The last thread arriving at the barrier provokes the generation of IPIs (Inter-Processor Interrupt) to all the other threads in a passive wait mode (to wake them up)
  * Sequential process managed by kernels
  * Our idea was to provide HW IP for multicast IPI generation
* Same application with 2 kinds of barriers (on a 64 cores arch.)
  * Pthread barrier (explicit barriers)
  * GNU OpenMP (implicit barriers)

	Median exec. time
Pthread app.	4882023 cycles
GNU OpenMP app.	5398377 cycles

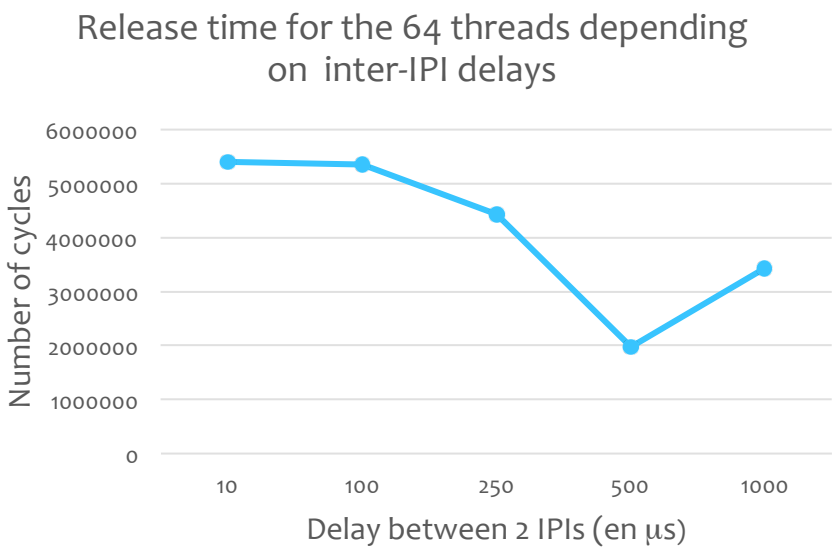
* For both applications, we have observed a ***memory contention***, as all threads on passive wait are awaked at the same time

Optimizations for passive wait

* The idea is to introduce a delay between 2 IPIs and to measure the best performance



Pthread



GNU OpenMP

Optimizations for passive wait

* The best release time is obtained with a 500 µs delay between 2 IPIs

inter-IPI delay	Median exec. time	Gain
Pthread app. ref	4882023 cycles	
500 µs	1932063 cycles	61 %

inter-IPI delay	Median exec. time	Gain
GNU OpenMP app. ref	5398377 cycles	
500 µs	1970235 cycles	64 %

* A lot of questions
  * How to get the optimal delay ?
  * Is it platform dependant ?
  * …

Conclusion and Perspectives

* A non-intrusive measurement tool chain for Accurate Analysis of Synchronization barrier
  * Improvement and validation:
    * Active wait: GNU OpenMP library
      * About 90% improvement of release time
    * Passive wait: GNU OpenMP and Pthread
      * Introduction of a delay between 2 IPIs for a 60% improvement of release time

* Next steps
  * Active wait:
    * Validation on multiprocessor machine and on simulation with some more applications
  * Passive wait:
    * How to determine a minimum delay to generate IPI ?

MPSoC '17

Accurate Study and Optimization of Synchronization Barriers in a NoC based MPSoC Architecture

Frédéric Rousseau

TIMA lab – University of Grenoble Alpes

A joint work with Maxime France-Pillois et Jérôme Martin
CEA LETI