

Multicore and Hard Real-Time Systems

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`

2009-09-04

What This Talk is About

Multicore gives rise to new problems for hard real-time systems

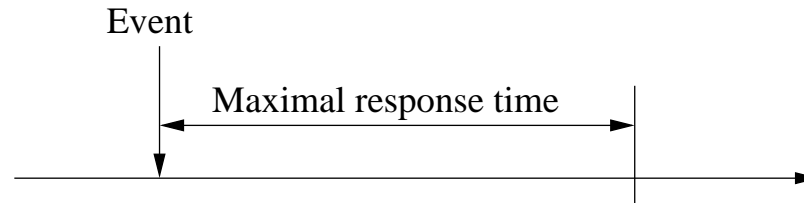
I will explain why, and discuss how the problems might be alleviated

Structure of talk:

- Short intro to real-time systems and Worst-Case Execution Time (WCET) analysis
- Why multicore is problematic for hard real-time systems
- Possible ways to alleviate the problem

Real-Time Systems

Real-Time Systems are systems with *timing constraints*



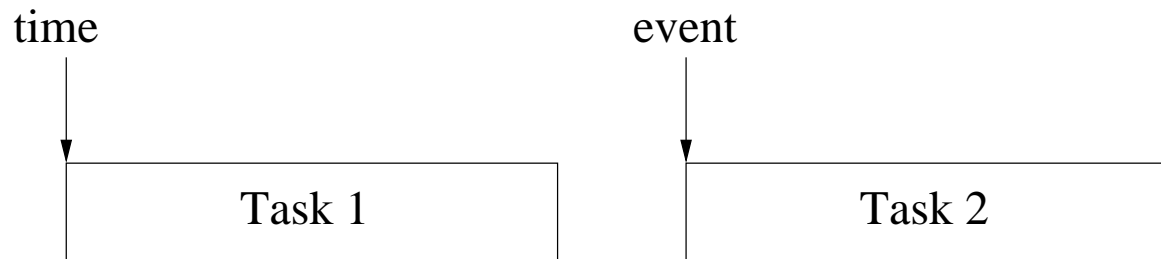
Absolute deadlines, maximum response times, bounds on execution time variation, etc.

Two classes:

- *Hard* real-time systems: timing constraints *must* be met (typically safety-critical systems)
- *Soft* real-time systems: *desirable* that timing constraints are met (but not absolutely necessary)

Task Model

Real-time system design, if done according to the book, uses a *task model*:



Tasks are short, terminating programs that usually recur

Triggered by *time* (often cyclically), or *events*

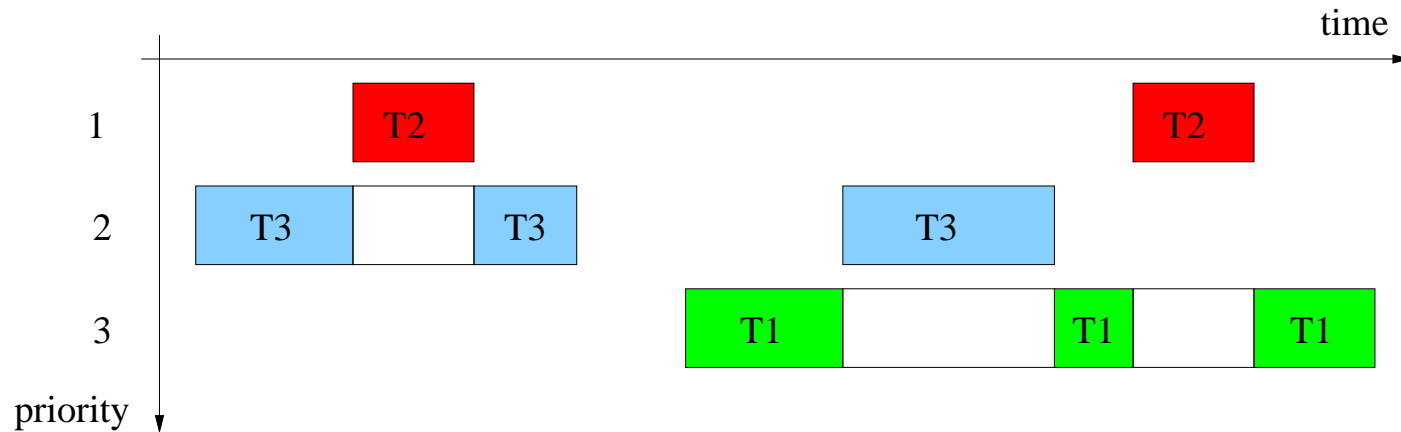
Task Scheduling

Tasks are run according to some *scheduling policy*, for instance:

Cyclically:



Priority-based:



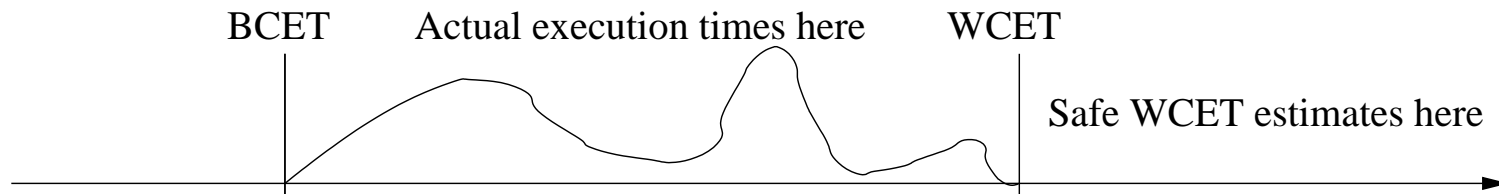
Analysis of Real-Time Properties

Systems of tasks, under a scheduling policy, can be *analyzed* w.r.t. real-time properties

E.g., maximal response time for a task in a system with priorities

There is a bulk of theory for analyzing task systems under different scheduling policies

However, tasks can have varying *execution times*



In particular, the analyses need safe (not underestimated) estimates of the *Worst-Case Execution Time (WCET)*

WCET Analysis

Worst-Case Execution Time (WCET) Analysis – find safe upper bounds to execution time of *sequential, uninterrupted* code

Common practice today:

- Perform extensive testing and record max execution time T
- Add a safety margin to T , and take that as WCET estimate

Usually not safe! Hard to know if the safety margin is sufficient

May want to use large safety margin to be on the safe side, may lead to large overestimations of WCET and thus poor resource utilization

A Better Way to Perform WCET Analysis

A mathematical analysis based on formal models is a safer alternative

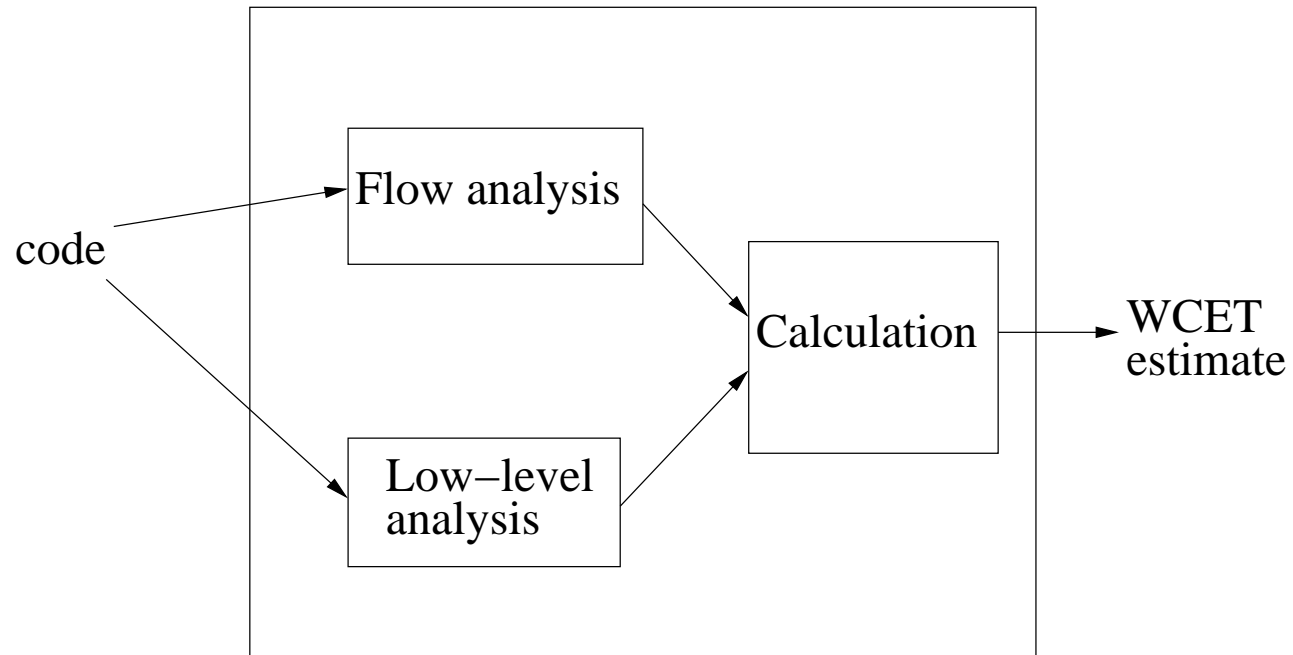
Variations in execution time can be caused by both software (different inputs \implies different paths through the code) and hardware (different hardware states give different execution speed for instructions)

Thus, a safe WCET analysis requires formal models of both hardware and software

Common way to estimate WCET:

1. find possible execution paths (*flow analysis*)
2. find timing for execution paths from hardware model (*low-level analysis*)
3. Find a safe estimate to the largest time (*calculation*)

Canonical Structure of WCET Analysis Tool



Low-Level Analysis

Safely estimate running time of small program pieces (typically basic blocks) given a set of possible initial HW states (cache contents, branch predictor state, etc)

This analysis can be very complex for modern HW architectures (caches, superscalar, . . .)

The CPU, but also **memory access times** etc. will influence running times

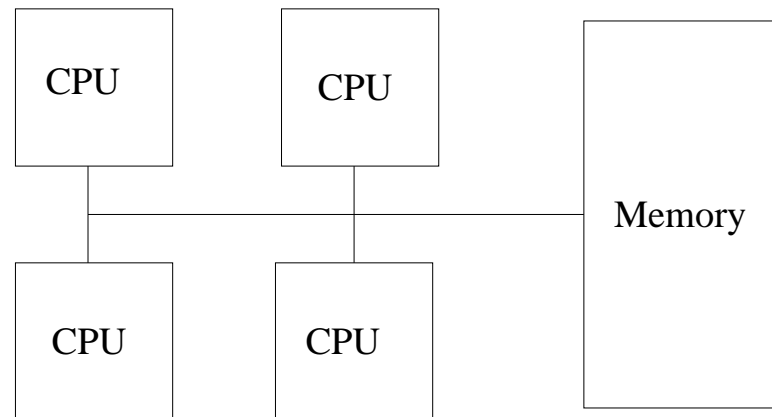
A lot of research efforts have gone into this, sophisticated analysis methods exist today

Yet, hard for timing analysis methods to keep up with hardware development

For multicore, things get even worse. . .

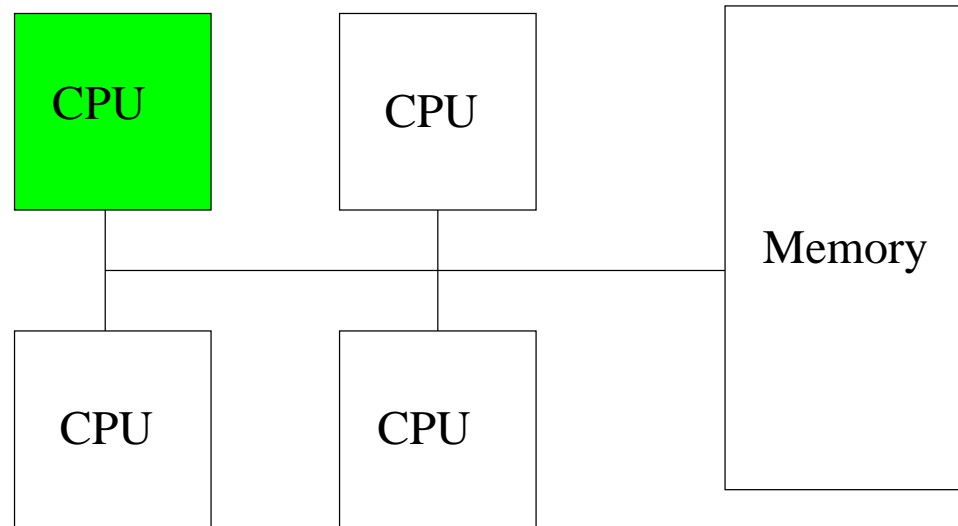
Multicore Processors

A vanilla multicore processor:

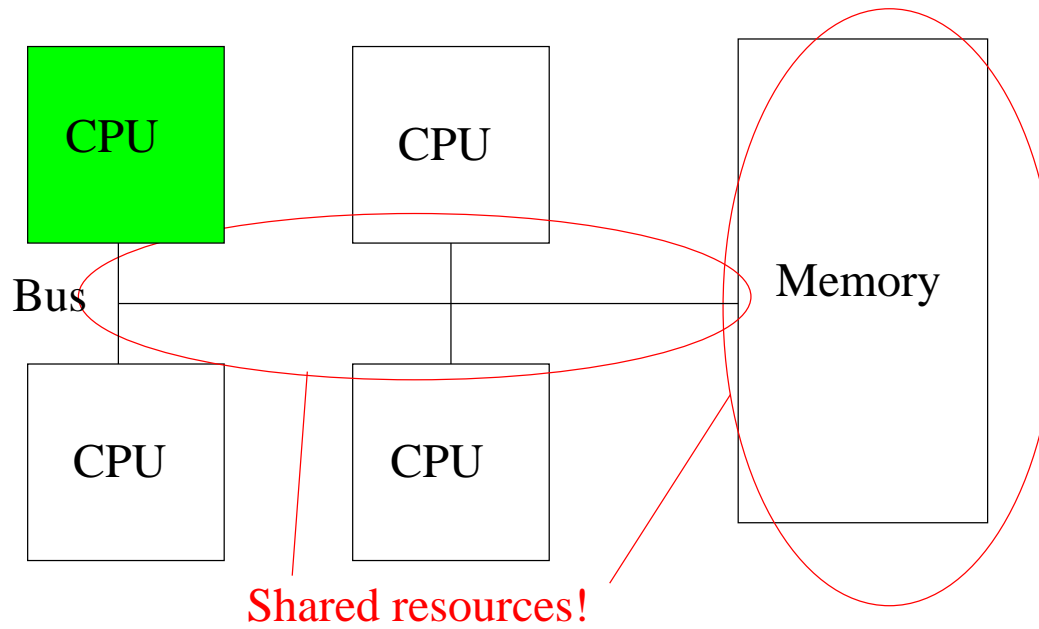


A number of processor cores, and a memory connected by a bus

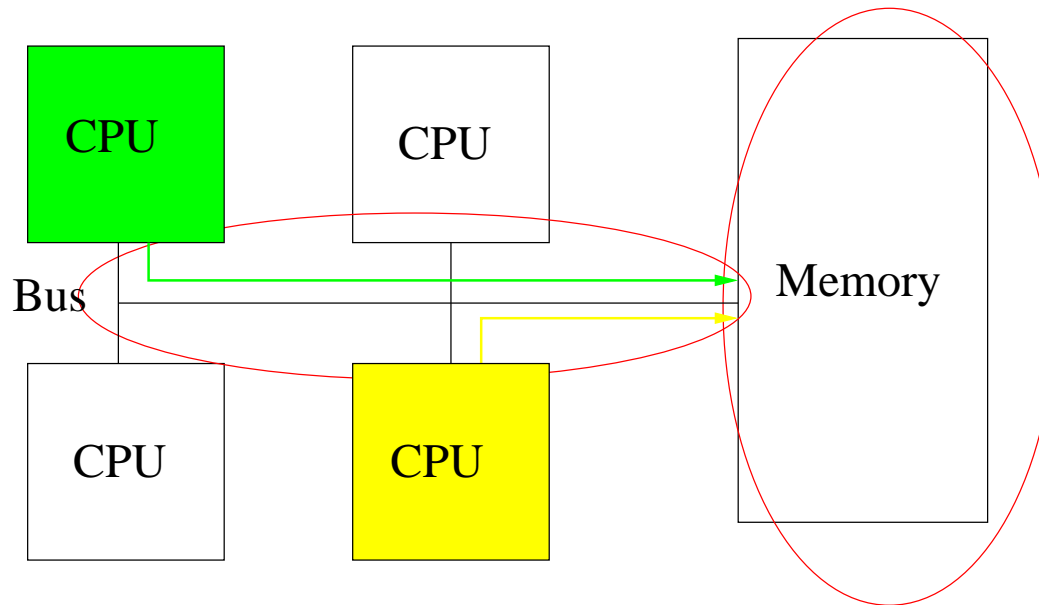
The Timing Analysis Problem for Multicore/MPSoC



Consider a task running on a core in a multicore system



The core will now *share resources* with other cores



When making a memory access, it will have to *compete* with the other cores for the shared resources needed

Memory access times thus become dependent on what is running on the other cores!

In the worst case, we cannot bound the memory access times. Then, estimated WCET = ∞

Basically two ways to bound the memory access times:

1. A detailed analysis of the memory accesses of all tasks running in the system
2. Some kind of access control to shared resources, which guarantees bounded access time (e.g., TDMA)

1 requires full information about all code running in the system \implies possible only for closed systems

Complexity grows fast

Won't work when memory references are too dynamic and unpredictable

2 requires HW support, will lower the utilization

A Possible Way Out

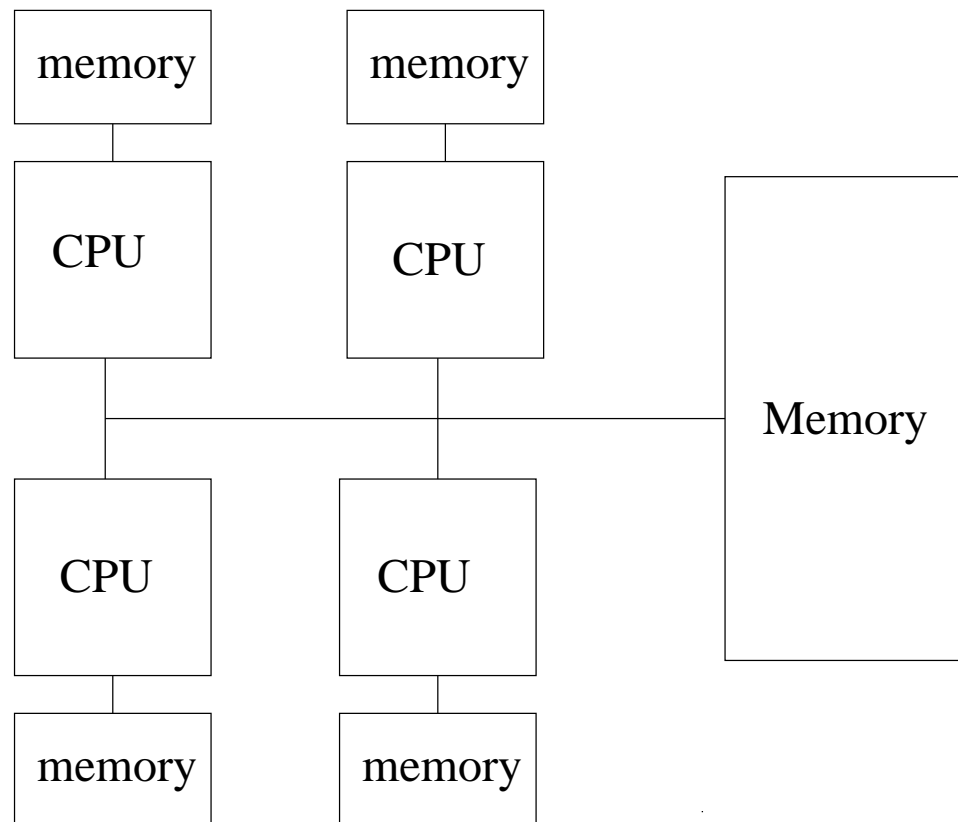
Consider also the software:

1. Find out which data can be made *private* (local to the task)
2. Put that data in local (private) memory
3. Let the rest of the data remain in shared memory
4. Put access to shared resources (bus/shared memory) under strict scheduling control

Privatization of data will reduce the pressure on shared memory, and reduce the penalty of putting bus/memory under scheduling control

Slogan: never share what you can keep to yourself!

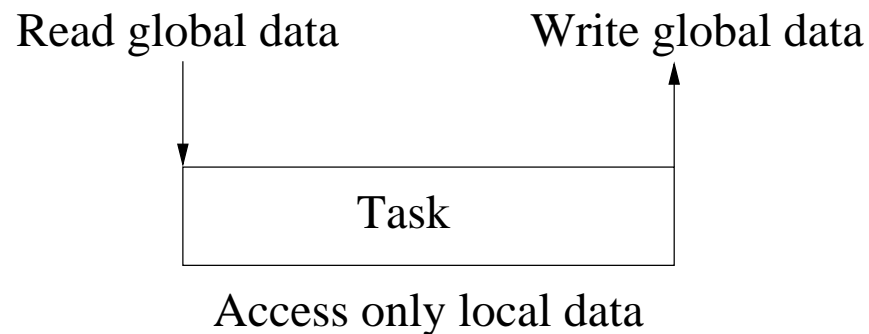
The approach requires that cores have local memories:



Preferrably scratchpads, caches will give problems to predict timing

A Clean Task Model

It helps to have a clean task model with controlled side-effects:

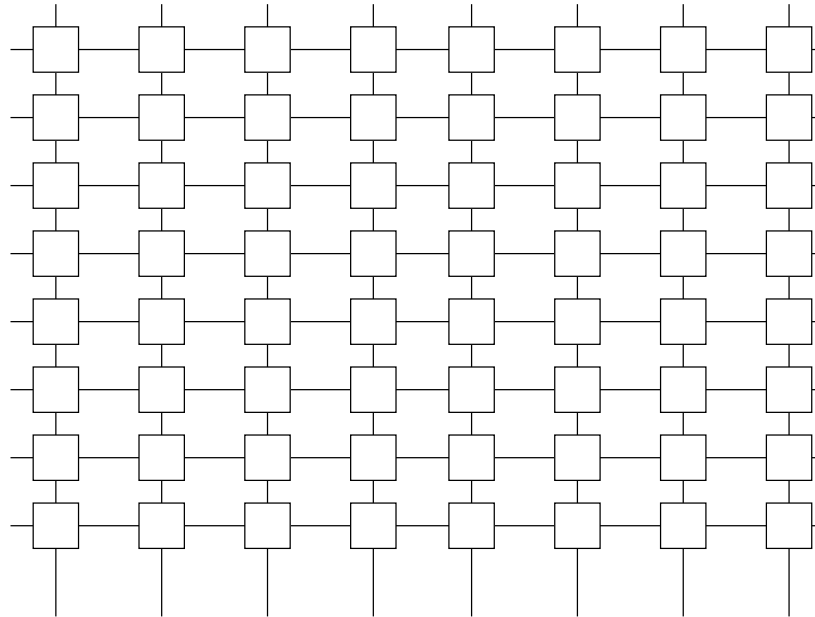


Makes it easier to predict timing effects of global memory accesses

Also ensures a clean functional semantics of the task (output pure function of input)

What About Future Manycore Architectures?

There is some reason to believe they might look like this:



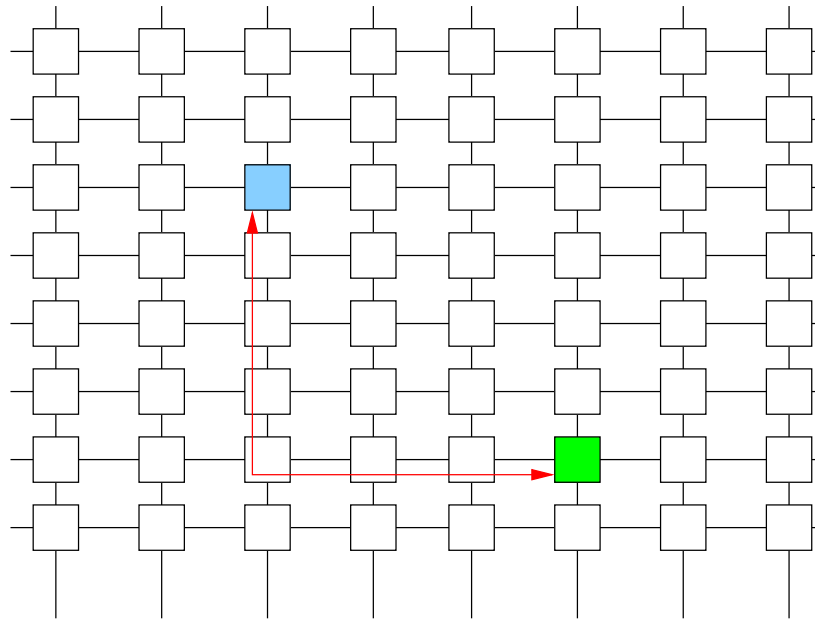
Massively parallel, distributed memory, Network on Chip (NoC)

How will this affect timing predictability?

May actually be beneficial if the design is done right:

- Distributed (local) memory is good for timing predictability
- A more distributed NoC (as compared to bus) should also be helpful, especially if communication can be localized
- Processor cores are likely to be less complex – will then have simpler timing models

The design of the NoC becomes crucial: it must be possible to guarantee bandwidth between different cores at certain times



Must be able to set up a route between two cores and obtain a guaranteed upper limit on the time to send a message

Conclusions

Multicore and hard real-time currently does not mix well

Shared resources yield large variations in execution times, and very unpredictable timing

Possible to alleviate these problems, but a performance penalty has to be paid

HW has to be designed for timing predictability – will HW vendors care?

SW likewise – will SW developers care?

If not, multicore will not be safe to use in time-critical systems