The Boot Process in Real-time Manycore Processors

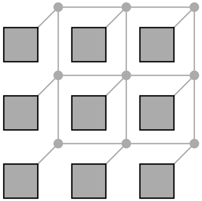
Florian Kluge, Mike Gerdes, Theo Ungerer

Department of Computer Science
University of Augsburg
Germany

RTNS 2014
October 8th, 2014

Manycore Processors

- Upcoming: Manycore processors for real-time domains
  - Kalray MPPA-256, Adapteva Epiphany
  - Cores connected by Network-on-Chip
  - Memory local to cores or clusters



- Opportunities for real-time domains:
  - Complex Algorithms

  → Decrease fuel consumption

  → New safety features

Challenge: Time-Predictability of Software

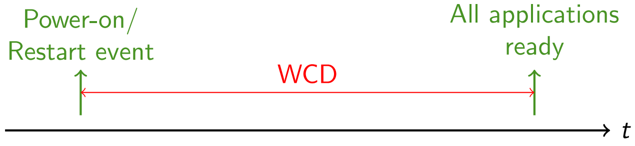
- Typically considered: regular operation
  - Assumes initialisation finished
  - Initialisation phase may be non-real-time!

- What about boot strapping?
  - Cold boot: loose constraints
  - Restart during operation due to fail-states, watchdog?
  - Fail-stop sometimes not possible

  $\rightarrow$ Bound blackout times!

Real-time Boot Process

- Boot process:
  - Each core needs memory image of code and data
  - Memory image: kernel + application image
  - Worst-case delay (WCD):

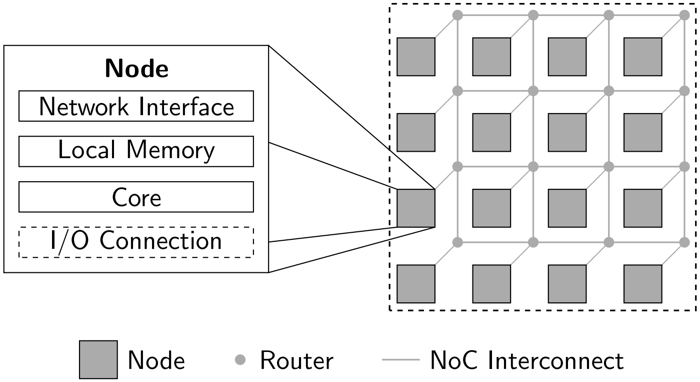


- Assumptions:
  - One core C0 has access to ROM
  - C0 coordinates boot process

- How should boot process be organised?
- Find a time bound for the WCD of the boot process!

Overview

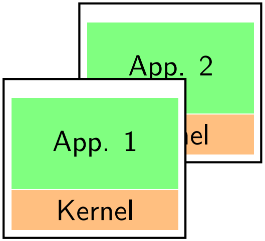
Target Platform

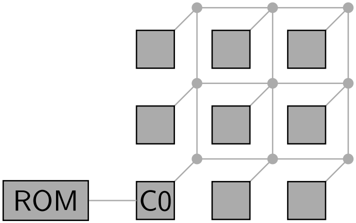


- Only few cores can access off-chip facilities (e.g. ROM, I/O)
- Communication based completely on explicitly sent messages

Full Image (FI)

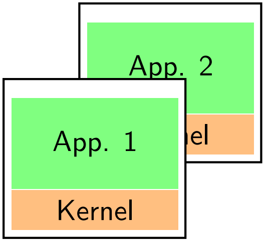
- One separate image for each core
- Contains kernel + core-specific application
- Everything loaded from ROM by C0

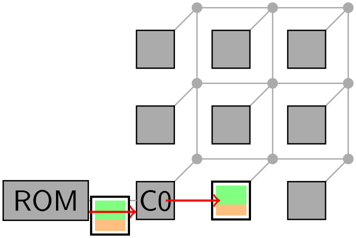




Full Image (FI)

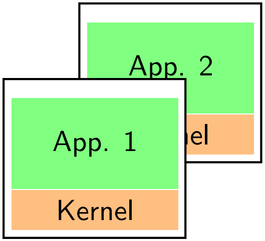
- One separate image for each core
- Contains kernel + core-specific application
- Everything loaded from ROM by C0

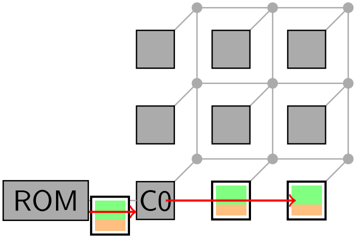




Full Image (FI)

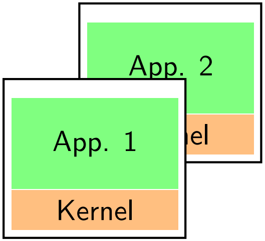
- One separate image for each core
- Contains kernel + core-specific application
- Everything loaded from ROM by C0

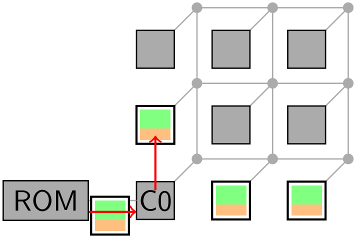




Full Image (FI)

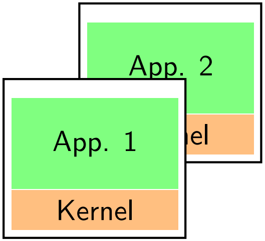
- One separate image for each core
- Contains kernel + core-specific application
- Everything loaded from ROM by C0

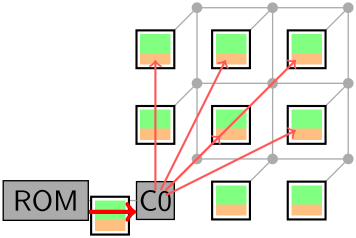




Full Image (FI)

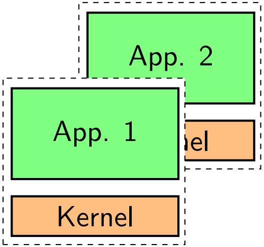
- One separate image for each core
- Contains kernel + core-specific application
- Everything loaded from ROM by C0

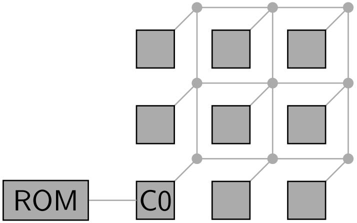




Split Image (SI)

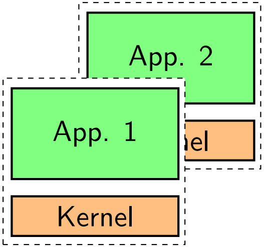
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

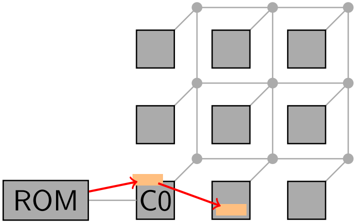




Split Image (SI)

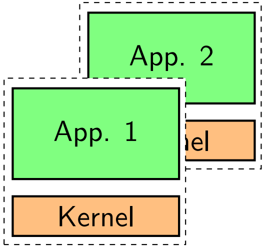
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

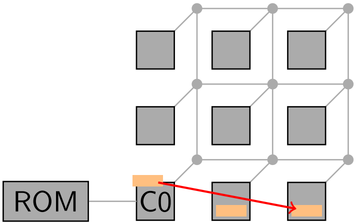




Split Image (SI)

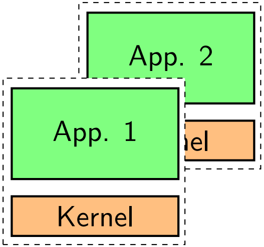
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

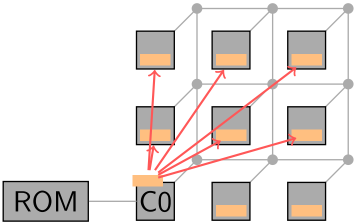




Split Image (SI)

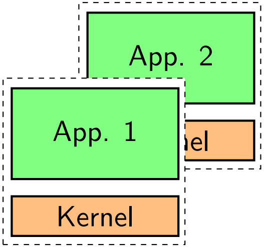
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

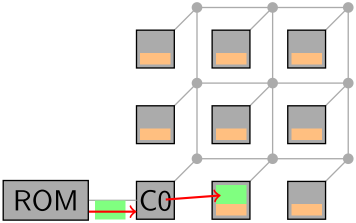




Split Image (SI)

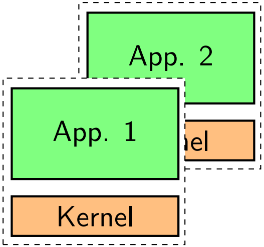
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

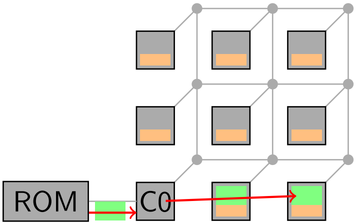




Split Image (SI)

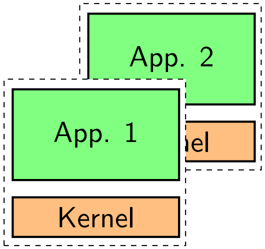
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

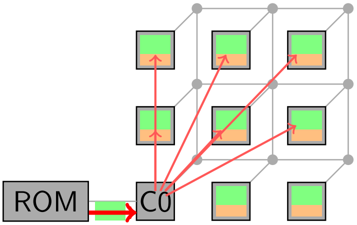




Split Image (SI)

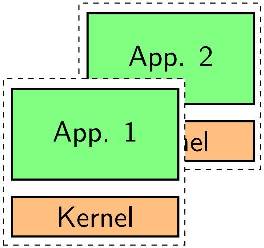
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to all cores
- Application image like in FI

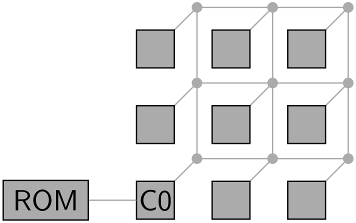




Self-Distributing Kernel

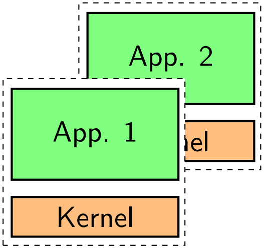
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI

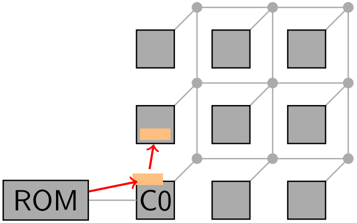




Self-Distributing Kernel

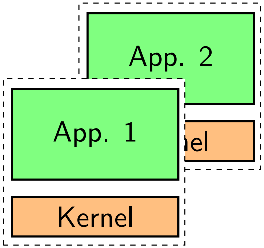
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI

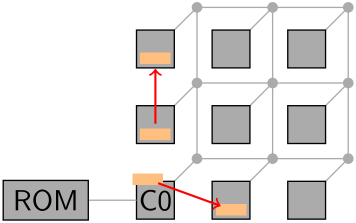




Self-Distributing Kernel

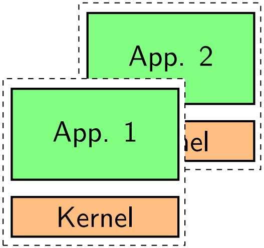
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI

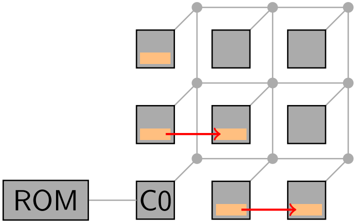




Self-Distributing Kernel

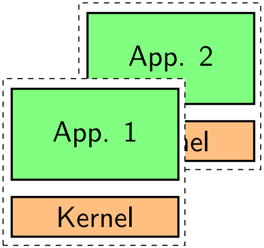
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI

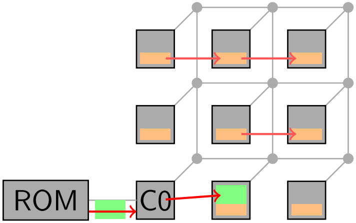




Self-Distributing Kernel

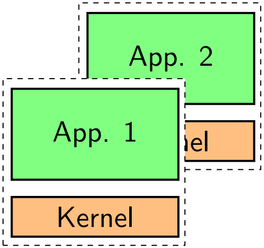
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI

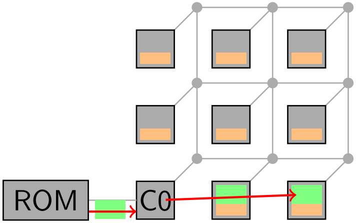




Self-Distributing Kernel

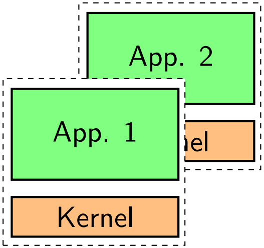
- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI

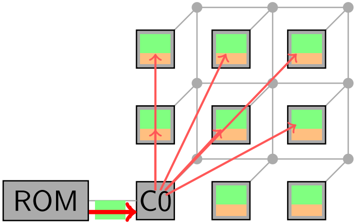




Self-Distributing Kernel

- Image split into generic kernel + core-specific application
- C0 loads kernel only once from ROM, sends to some cores
- Starting kernel distributes itself to other cores
- Application image like in FI/SI





Evaluation Methodology

- Abstract simulation of bootstrapping steps
  - Sequential execution
  - Communication between cores
- Assume worst-case timing for any step
- Abstract cores finish simulation at times $w_1, \ldots, w_n$

$\rightarrow$ Worst-Case Delay:

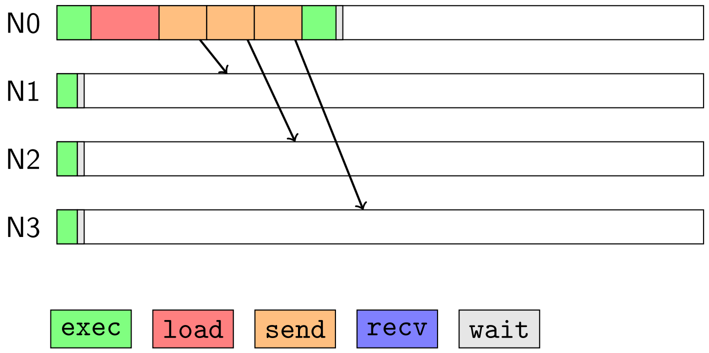
$$WCD = \max\{w_1, \ldots, w_n\}$$

MWSim Scripts

- Sequential execution for c cycles: `exec c`
  - $\rightarrow$ c retrieved with OTAWA
- Loading n bytes from ROM: `load n`
  - $\rightarrow$ $w = nl_{ROM}$, l_{ROM} via OTAWA and ROM timing
- Waiting for a message
  - $\rightarrow$ w depends on arrival time of message
- Sending/receiving messages
  - $\rightarrow$ w via OTAWA, message size, and network interface timing
- `foreach`-loop
  - $\rightarrow$ w: WCL of subprogram for each loop execution
- Wait for same message type from several cores: `parwait`
  - $\rightarrow$ w: WCL of subprogram for each loop execution plus waiting times

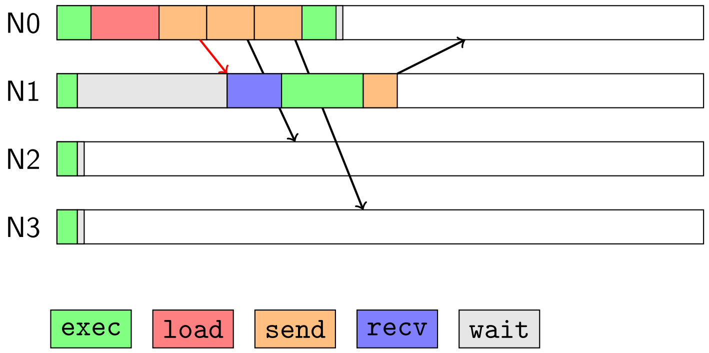
Script Execution

1. Each core executed until waiting point (`wait`/`parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



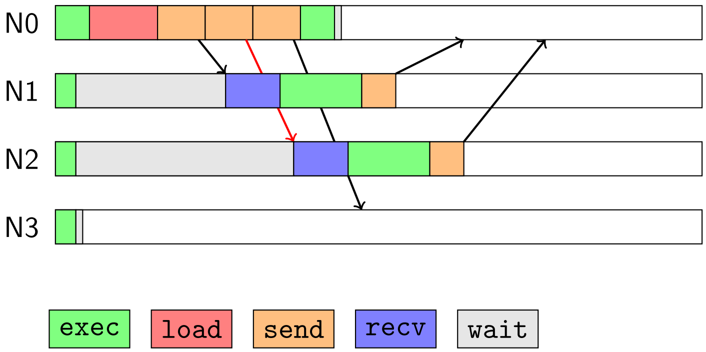
Script Execution

1. Each core executed until waiting point (`wait/parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



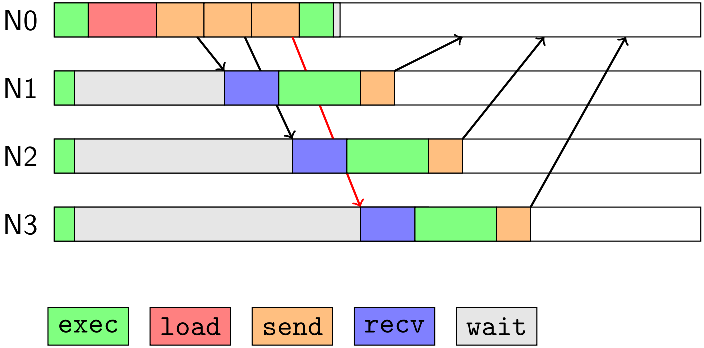
Script Execution

1. Each core executed until waiting point (`wait`/`parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



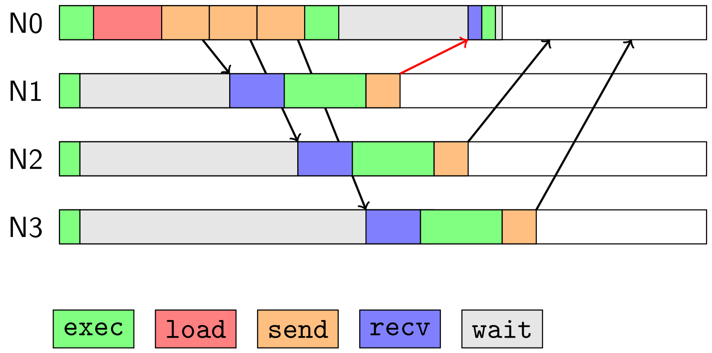
Script Execution

1. Each core executed until waiting point (`wait/parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



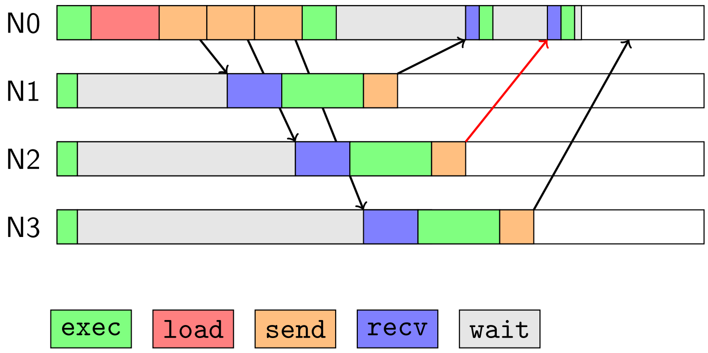
Script Execution

1. Each core executed until waiting point (`wait/parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



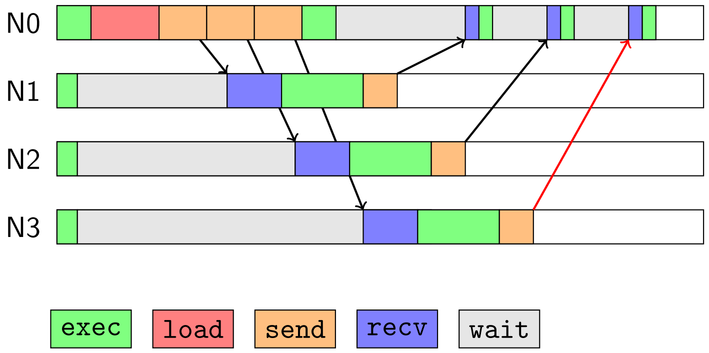
Script Execution

1. Each core executed until waiting point (`wait`/`parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



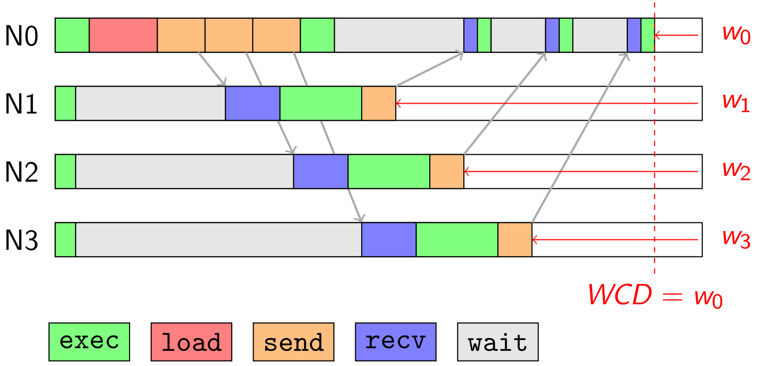
Script Execution

1. Each core executed until waiting point (`wait/parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages



Script Execution

1. Each core executed until waiting point (`wait/parwait`)
2. Deliver message with earliest arrival time
3. Execute receiver until next waiting point/end of script
4. Continue with 2. until no more messages

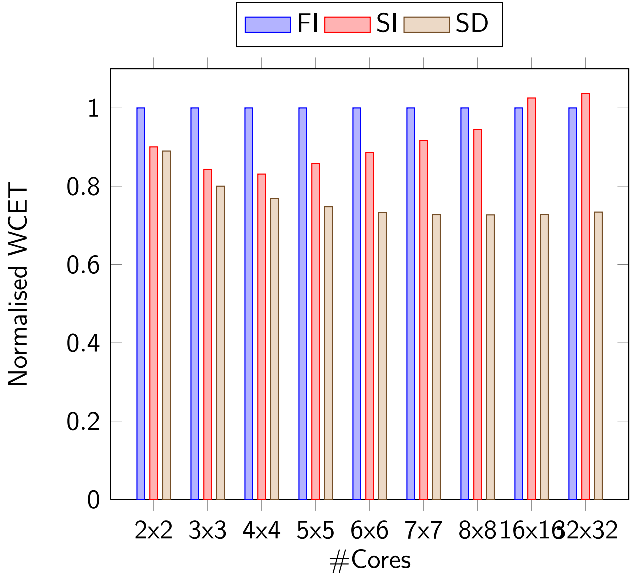


Evaluation Scenario

- Approaches implemented in MOSSCA
  - Manycore Operating System for Safety-Critical Applications
  - Research platform
  - Runs on manycore simulator
- Coordination performed by core 0
- Core Architecture: ARMv7, ARM Thumb ISA
- Mesh real-time NoC with fixed traversal times
- Chip sizes: 2x2 - 8x8, 16x16, 32x32 cores

Worst-Case Durations

Normalised to FI



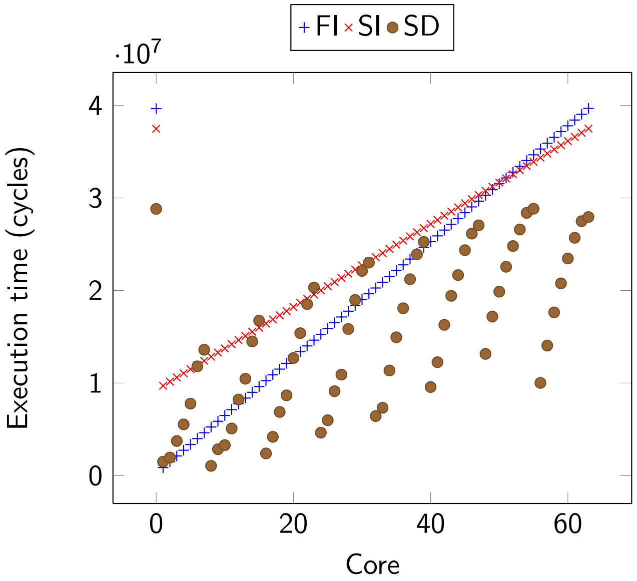
Worst-Case Durations

Values

Cores	FI	SI	SD
2x2	770,900	694,171	686,010
3x3	1,884,712	1,589,378	1,508,050
4x4	3,580,181	2,975,200	2,750,241
5x5	7,024,712	6,026,258	5,250,923
6x6	12,921,407	11,445,768	9,472,063
7x7	23,118,166	21,200,584	16,810,684
8x8	39,674,211	37,493,774	28,834,226
16x16	968,858,078	993,482,216	705,480,325
32x32	30,732,373,725	31,871,846,984	22,554,961,381

Experienced Durations

8x8 System



Summary

- WCD analysis of manycore boot process
  - Abstract simulation
  - MWSim Tool
- Three bootstrapping approaches
  - Full Image (FI)
  - Split Image (SI)
  - Self-Distributing kernel (SD)
- Results
  - SD up to 27% faster than FI
  - Large chips: SI can be slower than FI

Future Work

- Use DMA units for data transfer
- Extend analysis to other manycore architectures
  $\rightarrow$ Clusters of cores with cluster-memories
- Investigate usage of best-effort NoC
  $\rightarrow$ may reduce WCDs by two orders of magnitude
- Prioritisation of some nodes/applications, give guarantees to single nodes