Multithreaded Code from Synchronous Programs: Generating Software Pipelines for OpenMP

Daniel Baudisch, Jens Brandt, Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern, Germany

Motivation

1. Introduction
2. Creating Multi-Threaded Code
3. Results

Outline

1 Introduction

2 Creating Multi-Threaded Code

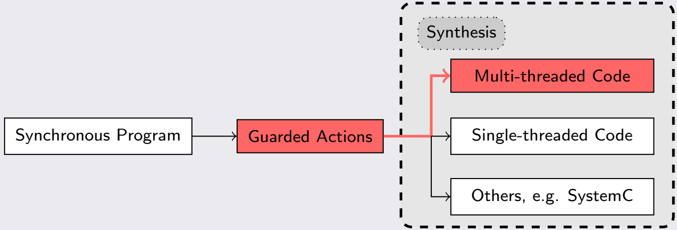
3 Results

Motivation

- synchronous languages, e. g. Esterel, Lustre, Quartz can be used in Embedded Systems
- generation of single threaded code so far
- multicore processors more frequently used
  - adapt SW synthesis ⇒ generation of multithreaded code

Synthesis Flow

Synthesis Flow



- synchronous languages ⇒ see talk of Mike Gemünde
- here: from synchronous guarded actions to multi-threaded code using OpenMP

Guarded Actions

System (Example)

Interface:

- Inputs: i, c
- Output: o
- Locals: x,y,z

Guarded Actions:

$$\begin{aligned} c &\Rightarrow o = x + y \\ \text{true} &\Rightarrow x = i \cdot i \\ \text{true} &\Rightarrow z = 2 \cdot i \\ \text{true} &\Rightarrow \text{next}(y) = z + 1 \end{aligned}$$

Guarded Actions

System (Example)

Interface:

Inputs: i, c

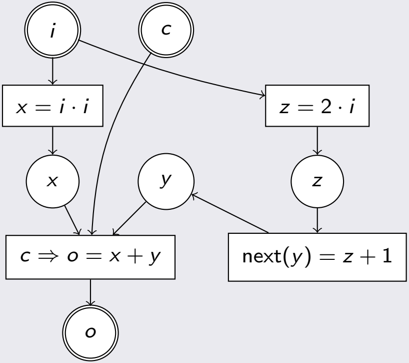
Output: o

Locals: x,y,z

Guarded Actions:

$$\begin{aligned} c &\Rightarrow o = x + y \\ \text{true} &\Rightarrow x = i \cdot i \\ \text{true} &\Rightarrow z = 2 \cdot i \\ \text{true} &\Rightarrow \text{next}(y) = z + 1 \end{aligned}$$

Dependency Graph



Guarded Actions

System (Example)

Interface:

Inputs: i, c

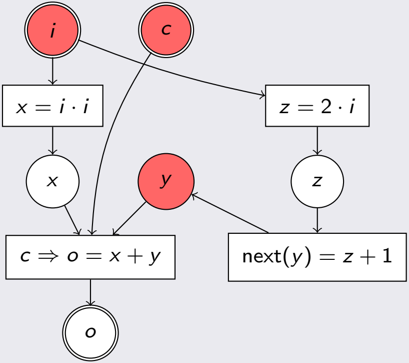
Output: o

Locals: x,y,z

Guarded Actions:

$$\begin{array}{rcl} c & \Rightarrow & o = x + y \\ \text{true} & \Rightarrow & x = i \cdot i \\ \text{true} & \Rightarrow & z = 2 \cdot i \\ \text{true} & \Rightarrow & \text{next}(y) = z + 1 \end{array}$$

Dependency Graph



Guarded Actions

System (Example)

Interface:

Inputs: i, c

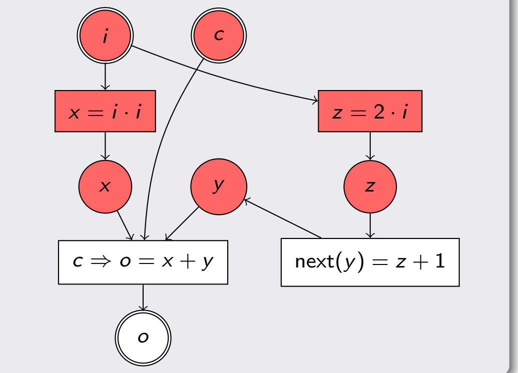
Output: o

Locals: x,y,z

Guarded Actions:

$$\begin{array}{rcl} c & \Rightarrow & o = x + y \\ \text{true} & \Rightarrow & x = i \cdot i \\ \text{true} & \Rightarrow & z = 2 \cdot i \\ \text{true} & \Rightarrow & \text{next}(y) = z + 1 \end{array}$$

Dependency Graph



Guarded Actions

System (Example)

Interface:

Inputs: i, c

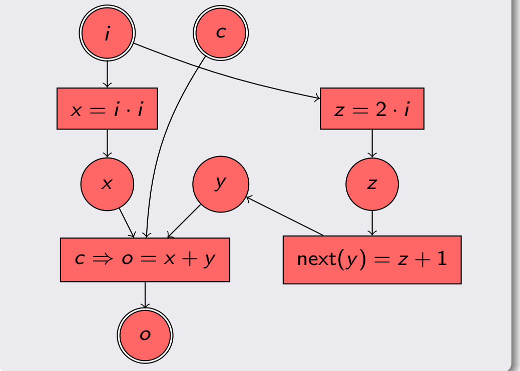
Output: o

Locals: x,y,z

Guarded Actions:

$$\begin{aligned} c &\Rightarrow o = x + y \\ \text{true} &\Rightarrow x = i \cdot i \\ \text{true} &\Rightarrow z = 2 \cdot i \\ \text{true} &\Rightarrow \text{next}(y) = z + 1 \end{aligned}$$

Dependency Graph



Outline

1 Introduction

2 Creating Multi-Threaded Code

3 Results

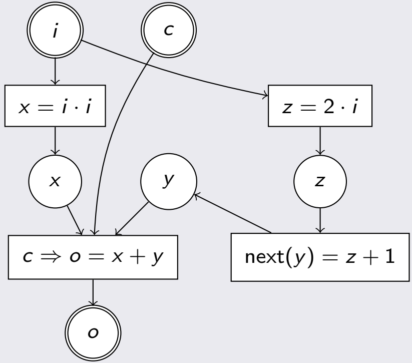
Extracting Independent Threads

First Approach

intuitive:

- group dependent actions: create "vertical slices"
- execute groups in parallel

Dependency Graph



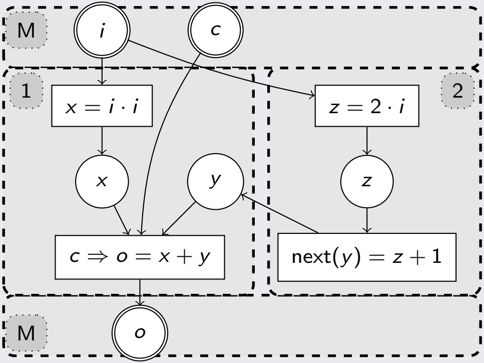
Extracting Independent Threads

First Approach

intuitive:

- group dependent actions: create "vertical slices"
- execute groups in parallel

Dependency Graph Multithreaded



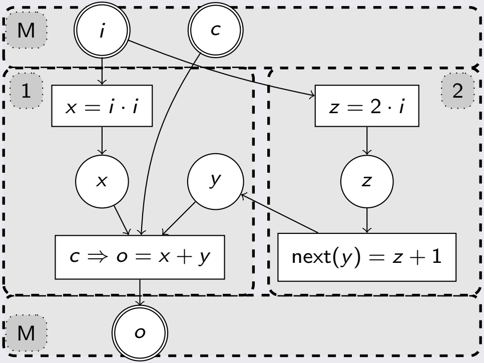
Extracting Independent Threads

First Approach

intuitive:

- group dependent actions: create "vertical slices"
- execute groups in parallel
- due to sync. overhead: only applicable for large groups
- problem: what if creation of large groups fails?

Dependency Graph Multithreaded



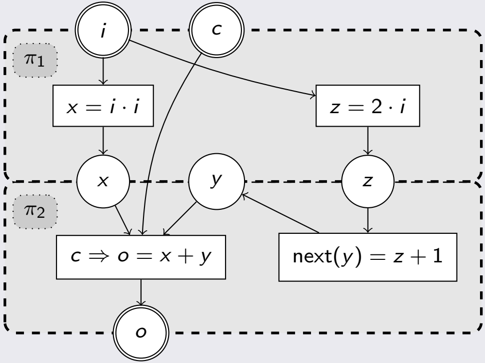
Pipelining

Second Approach

pipelining:

- group dependent actions: create "horizontal slices"
- basic idea: execute groups in parallel like a pipeline

Example - Pipelining



Pipelining

Problems:

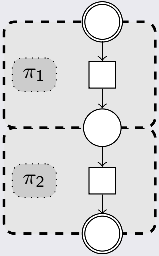
- How to partition dependency graph ?
  - ⇒ use a legal partitioning
  - ⇒ optimal partition depends on target architecture (not goal of this approach)
- What about values that are read in several stages ?
  ⇒ insert intermediate variables
- How to store intermediate values ?
  ⇒ use queues
- Where to write values at ?
- Do we require something like stalling ?
  ⇒ implicit by queues

Pipelining - Legal Partitioning

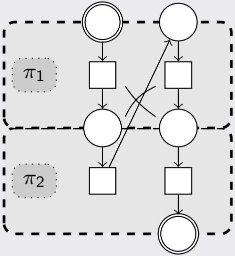
A partitioning is legal iff

- ctrl/data flow goes to one direction
- NOTE: a delayed write access may go to a previous stage

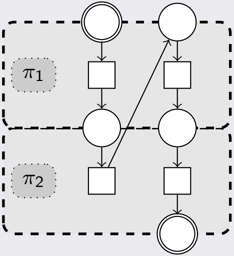
Legal



Not-Legal



Legal

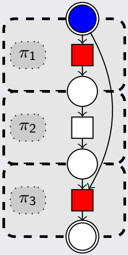


Pipelining - Intermediate Variables

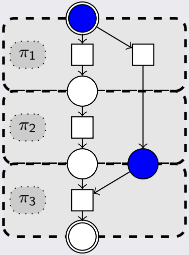
Insertion of Intermediate Variables (IV)

- copies of variables (comparable to pipeline register)
- implemented by queues
- whenever a variable is read by a stage

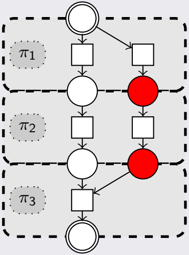
w/o IVs



IV inserted



IV in HW

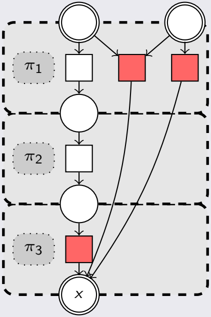


Pipelining - Write Access

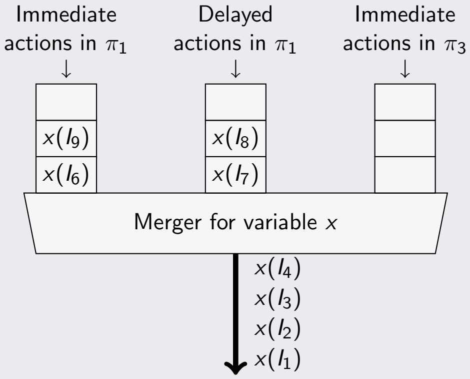
- forward write accesses to first intermediate variable (in spatial dimension)
  - ≈ forwarding in hardware
  - ⇒ order values using merge-element

Pipelining - Merge Element

Dependency Graph



Merger for x



Immediate actions in π_1

$x(l_9)$
$x(l_6)$

Delayed actions in π_1

$x(l_8)$
$x(l_7)$

Immediate actions in π_3

Merger for variable x

$x(l_4)$
$x(l_3)$
$x(l_2)$
$x(l_1)$

Translation to C

- using OpenMP: API for programming MT
- create thread for each stage
- each stage is executed in an own loop ⇒ allows stages to run desynchronized

Outline

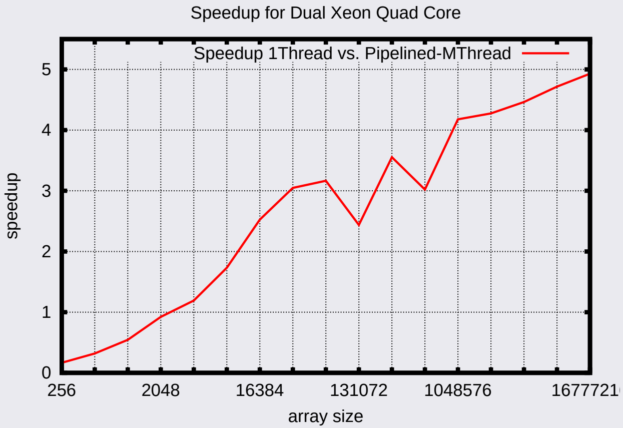
1 Introduction

2 Creating Multi-Threaded Code

3 Results

Benchmark

MergeSort - 2x Xeon Quad Core



The End

Thank you for your attention!

Questions? Suggestions? Ideas?