Synthesis of Synchronous Programs for Parallel Architectures

Daniel Baudisch

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

Outline

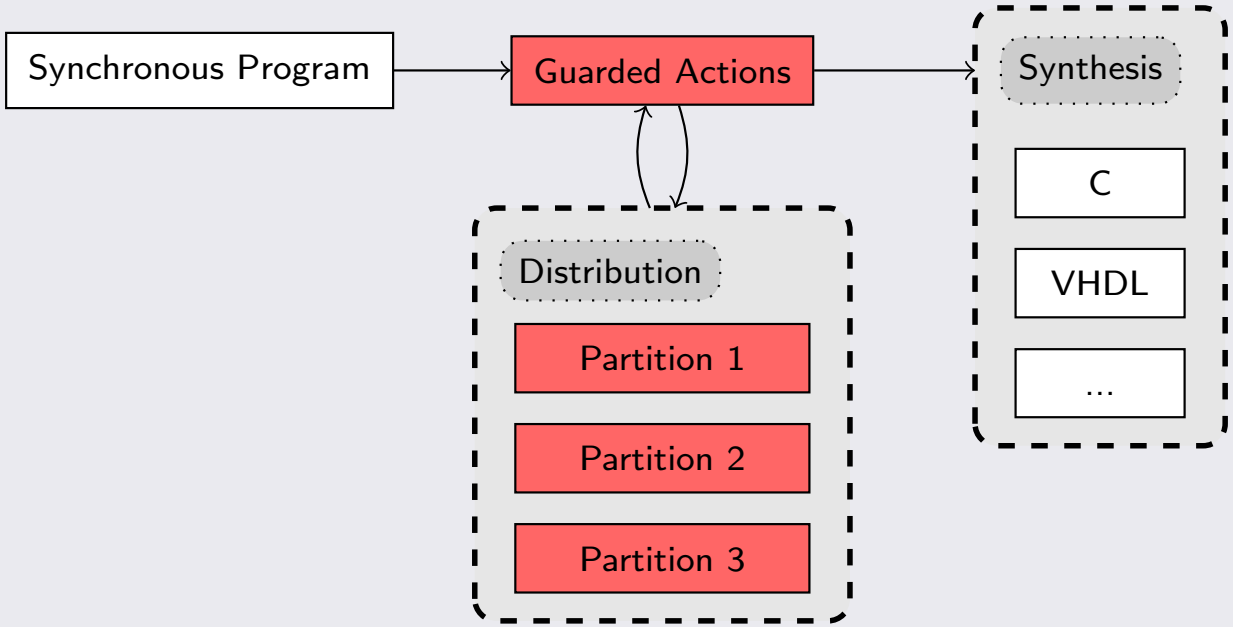
Outline

Motivation

- synthesis to sequential languages already given, e. g.
  - *Edwards*: Compiling Esterel into sequential code
  - *Weil et al*: Efficient Compilation of Esterel for Real-Time Embedded Systems
- synthesis to multithreaded code more challenging (especially for heterogenous/distributed systems)
- goal: enhancement of throughput

Synthesis Flow

Synthesis Flow



- here: from synchronous guarded actions to distributed systems

Guarded Actions

- intermediate format for synchronous languages
- same MoC as source language

Guarded Actions

System (Example)

Interface:

Inputs: i, c

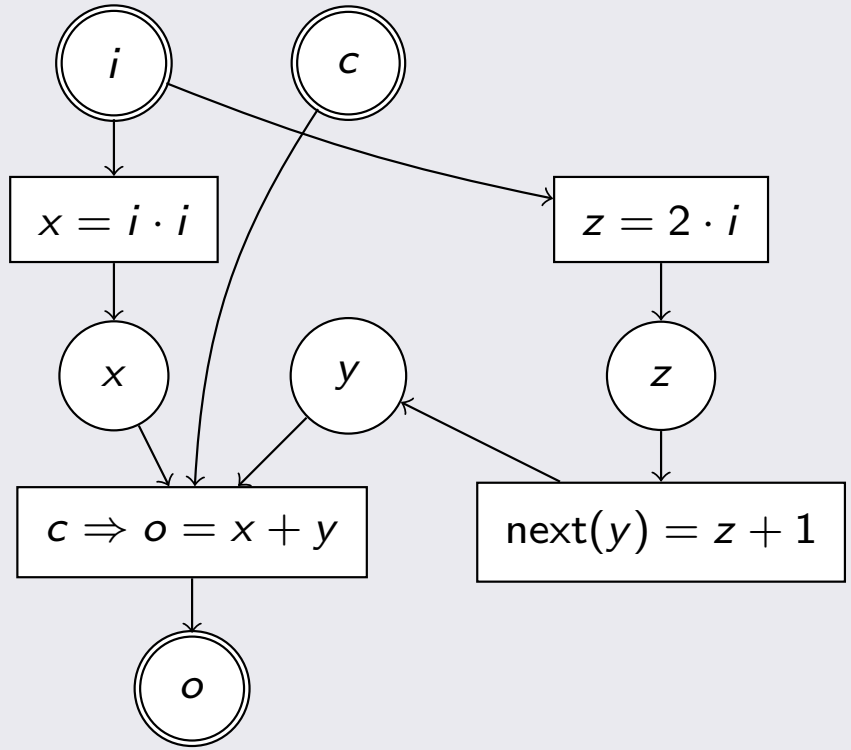
Output: o

Locals: x,y,z

Guarded Actions:

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

Dependency Graph (DG)



Creating Threads

Recent approaches partition DG:

- "vertical" slicing $\Rightarrow$ multiple threads to execute one step
- "horizontal" slicing $\Rightarrow$ pipelining of DG
- in progress: out-of-order execution
  $\Rightarrow$ applying techniques known from processor design

Outline

Approach

Basic idea:

- group actions
- avoid dependencies between groups
- non-depending groups can be run in parallel

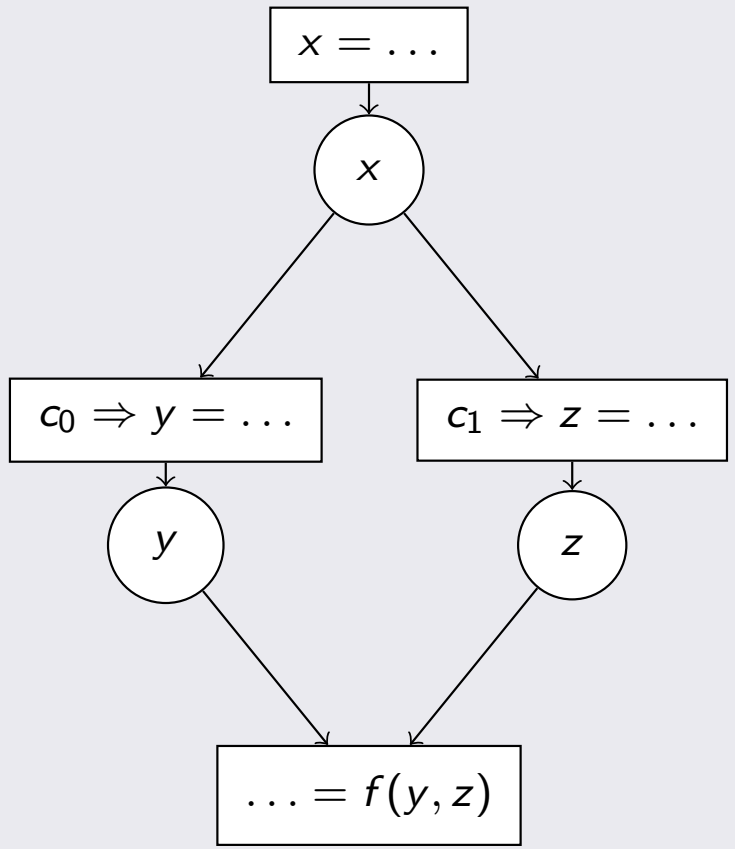
Approach

Insertion of Forks and Joins

Inserting pairs of *forks* and *joins* into the DG. In principle:

- fork, if a variable is used by two or more actions
- join, if an action depends on two ore more variables

DG



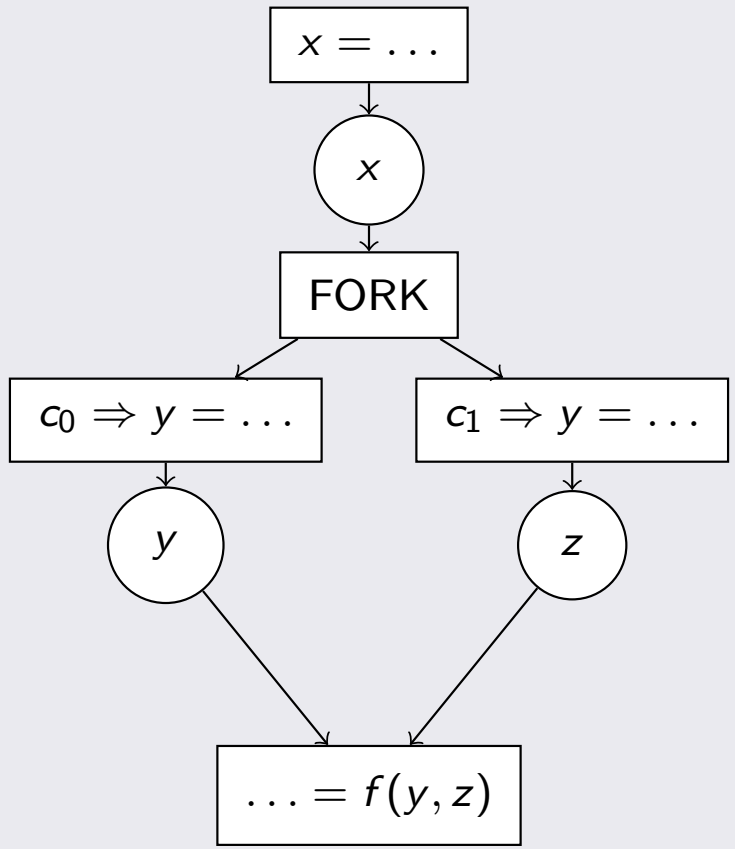
Approach

Insertion of Forks and Joins

Inserting pairs of *forks* and *joins* into the DG. In principle:

- fork, if a variable is used by two or more actions
- join, if an action depends on two ore more variables

DG



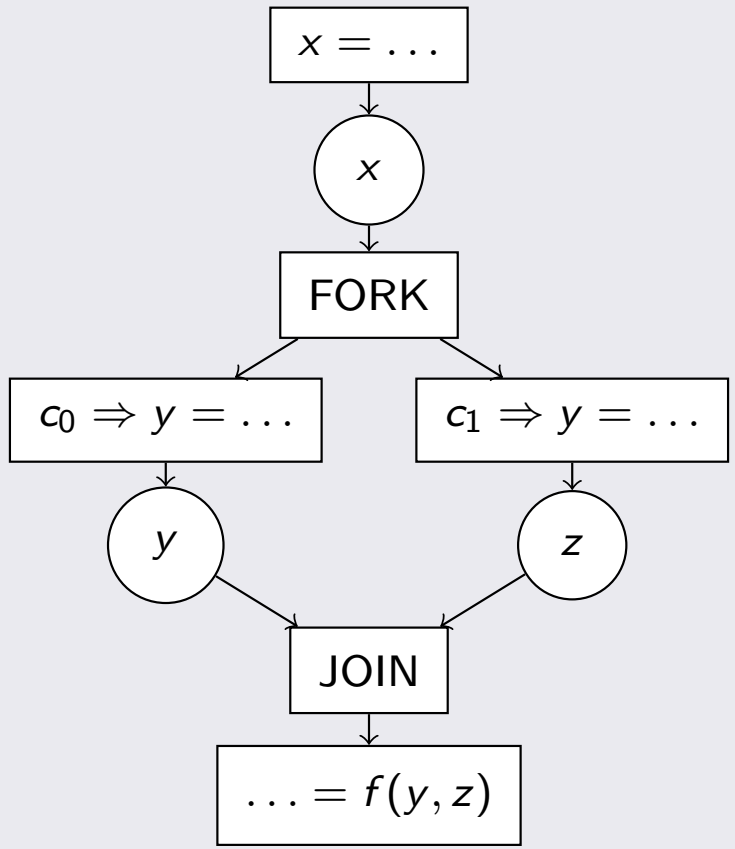
Approach

Insertion of Forks and Joins

Inserting pairs of *forks* and *joins* into the DG. In principle:

- fork, if a variable is used by two or more actions
- join, if an action depends on two ore more variables

DG



Approach

- each fork-join-pair encloses a set of threads
- fork-join-pairs can be nested $\Rightarrow$ nested parallelism
- can be synthesized, e. g. to C using OpenMP
  (fork-join-pairs must not overlap)
- details can be found in *Baudisch, Brandt, Schneider*:
  Multithreaded Code from Synchronous Languages: Extracting Independent Threads for OpenMP

Outline

Motivation

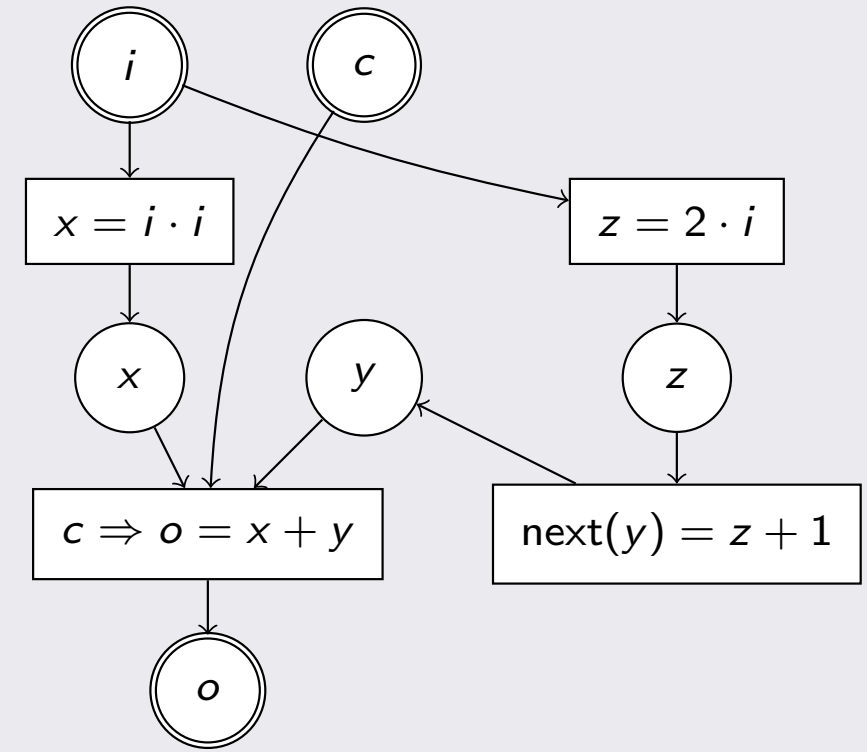
First approach may fail to create enough threads due to dependencies.

Approach

- break DG into components, such that
  - inputs of DG are inputs of first component
  - output one component is input of next component
  - outputs of last component are also outputs of DG
- each component is synthesized as one thread
- components can run asynchronously (GALS)
- data transfer between components done using fifo buffers $\Rightarrow$ TODO: reduction of transfer using *endochrony/isochrony*
- details can be found in *Baudisch*, *Brandt*, *Schneider*: Multithreaded Code from Synchronous Languages: Generating Software Pipelines for OpenMP

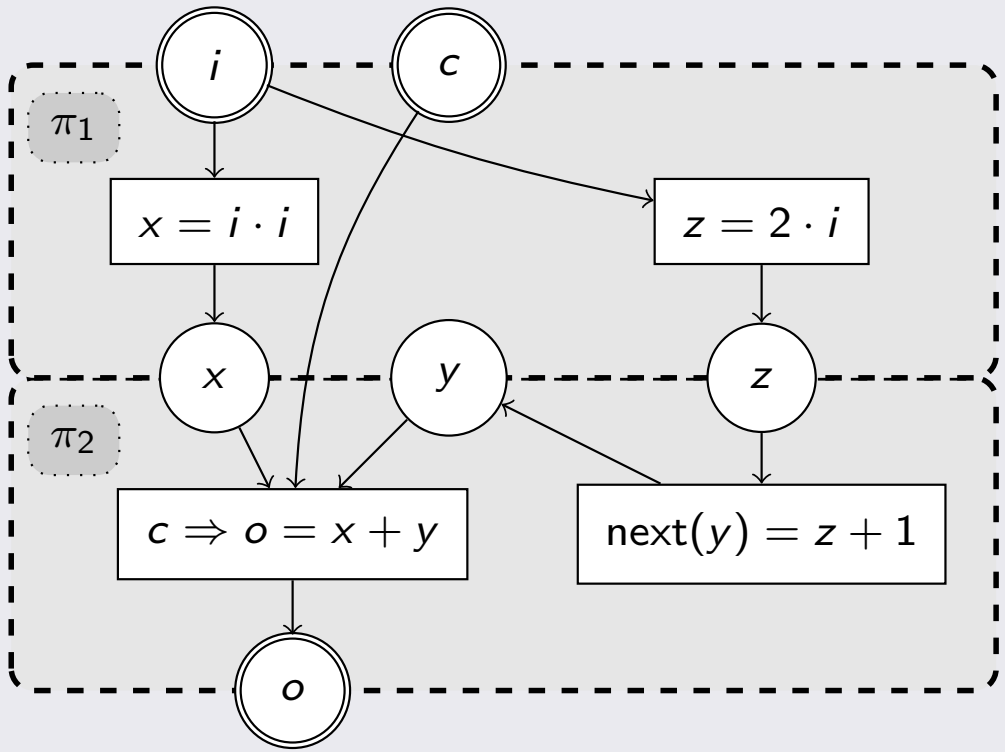
Example

DG



Example

Pipelined DG



Pros and Cons

- does not accelerate processing of one input set
- increases throughput
- same problems as in hardware design:
  data conflicts, e. g. RAW conflicts
  $\Rightarrow$ solved by using fifo buffers but have same effect as forwarding and stalling

Outline

Approach

Dynamic Scheduling + Data Flow Processing

- one table containing all inputs and intermediate results of input set $\Rightarrow$ comparable to reservation station + reorder buffer (RSRB)
- 3 threads to manage execution
  - reader thread
  - dispatcher thread
  - writer thread
- arbitrary number of threads to execute synchronous program

Approach

- synchronous program is translated to an arbitrary number of threads (components)
  - each component requires that (not necessary all but) some inputs and some local variables are known
  - a component should be executed for an input set as soon as these variables are known

  comparable to functional units in a processor's EX-stage
  but
  - software: apply each unit / input set
  - hardware: apply exactly one unit / input set

Approach

- one reader thread
  - reads inputs and puts them to the RSRB
- one dispatcher thread
  as soon as an entry for an input set changes:
  - compare available variables with those that are necessary to fire components
  - compare if all outputs are available and send values to writer thread
  - check if all components have been fired and remove input set
- one writer thread
  - send output values in-order to environment

Pros and Cons

- does not accelerate processing of one input set
- increases throughput
- analogous elegant resolving of data conflicts as in hardware
- out-of-order requires independency of input sets, or:
  TODO: *speculative execution*
  - causes race conditions
  - requires good speculations or
  - much much more cores (e. g. GPGPUs)

The End

Thank you for your attention!

Questions? Suggestions? Ideas?