Critical-Path-First Based Allocation of Real-Time Streaming Applications on 2D Mesh-Type Multi-Cores

Hazem Ismail Ali[1] Luís Miguel Pinho[1] Benny Akesson[2]

[1]CISTER Research Centre/INESC-TEC, Polytechnic Institute of Porto, Portugal

[2]Eindhoven University of Technology, The Netherlands

{*haali, lmp*}*@isep.ipp.pt, k.b.akesson@tue.nl*

RTCSA 2013 - Taipei - Taiwan

August 21, 2013

Overview

Introduction (1/4)

- Multi-core architectures integrating several low-performance cores on a single chip became popular.
- Streaming multimedia applications are becoming increasingly important and widespread.
- They have **high processing requirements** and **timing constraints** that must be satisfied, e.g., H.264 video decoders.
- The dataflow computational model is suitable for representing streaming applications because:
  1. it enables them to use the massive computational power of multi-core systems (**parallelization model**).
  2. it is a **natural paradigm** for representing them.
- A dataflow model is specified by a directed graph, where the nodes are considered as actors and the connections between the nodes, i.e. edges, as channels of data.

Introduction (2/4)

Homogeneous Synchronous Dataflow (HSDF)

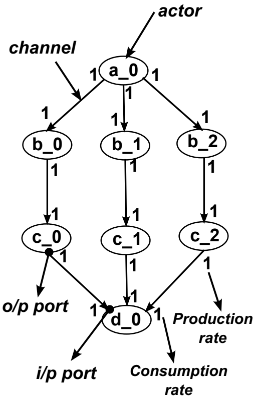


Figure : An example HSDF graph.

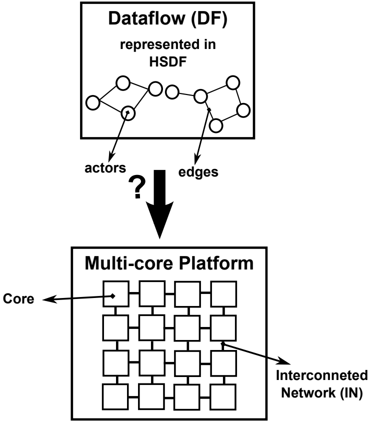
- Is a special case of dataflow graphs in which all rates (production / consumption) associated with actor ports are equal to 1.
- When each actor is fired once, the distribution of tokens on all channels return to their initial state (*complete cycle* or *graph iteration*).
- Other models (e.g. SDF, CSDF) can be converted to an equivalent HSDF using a conversion algorithm.

Introduction (3/4)

Problem

Problem to be addressed:

How to *Allocate* real-time streaming applications modeled as HSDF on a multi-core platforms such that we can guarantee satisfying its timing constraints?



Introduction (4/4)

- This allocation problem has previously been tackled in several works from a high-performance point-of-view. However, these approaches do not consider timing constraints and thus cannot be used for allocation of real-time dataflow applications.
- We propose a new algorithm called Critical Path First (CPF).
- CPF is for allocation of real-time applications modeled as HSDF dataflow graphs on 2D mesh multi-core processors.
- Results show that the proposed heuristic improves utilization of system resources with up to 7% and speeds up the allocation process with up to 19% compared to approaches using a First-Fit bin-packing heuristic.

System Model

Formally, we consider a system S based on :

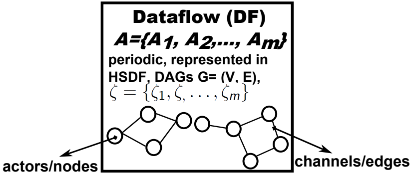


Figure : System Model.

System Model

Formally, we consider a system S based on :

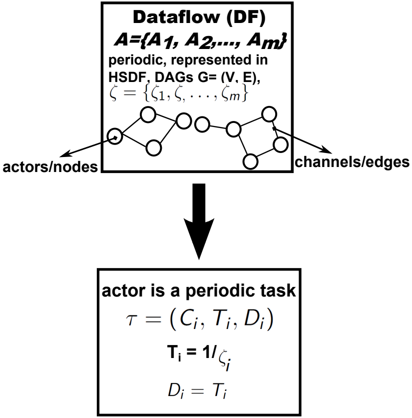


Figure : System Model.

System Model

Formally, we consider a system S based on :

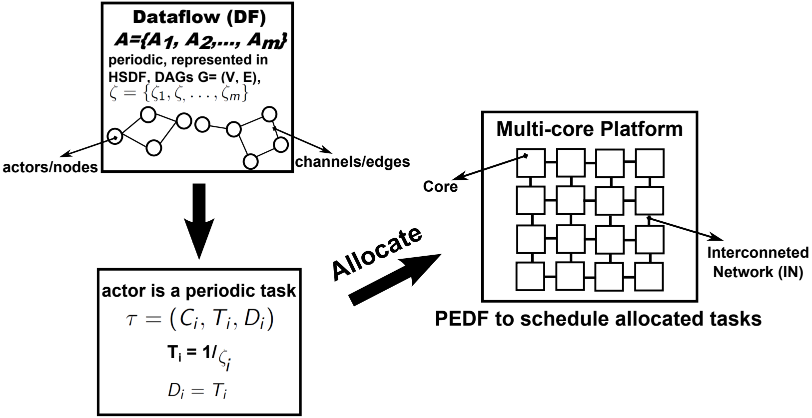


Figure : System Model.

Allocation Algorithm

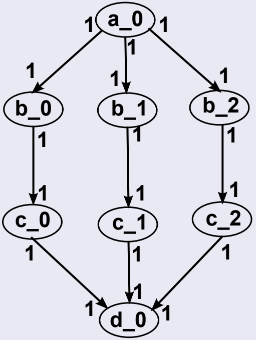
The algorithm is intended for allocation of applications modeled as HSDF graphs onto 2D mesh multi-cores at design time.

It consists of two main phases:

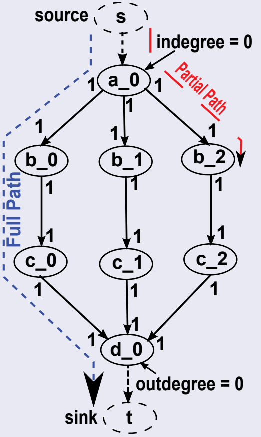
1. Finding all the possible paths between the nodes of the applications on the system.
2. Allocating the actors of the graph on the cores of the mesh processor using the output information of the previous phase.

First phase: Finding all possible paths

1) Creation of source and sink actors:



(a) An example HSDF graph.



(b) Adding source s and sink t.

First phase: Finding all possible paths

2) Path enumeration:

Partial Path :

$$P_i = \langle s = v_0, v_1, \ldots, v_j \rangle$$

Extend Partial Path using

$$Succ(v_j) = (v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_l})$$

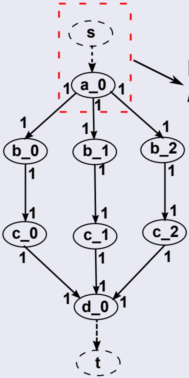
Resulted Paths :

$$\left.\begin{array}{l} P_{i_1} = \langle s = v_0, v_1, \ldots, v_j, v_{j_1} \rangle \\ P_{i_2} = \langle s = v_0, v_1, \ldots, v_j, v_{j_2} \rangle \\ \vdots \\ P_{i_l} = \langle s = v_0, v_1, \ldots, v_j, v_{j_l} \rangle \end{array}\right\}$$

PATHS	**Delay**

First phase: Finding all possible paths

2) Path enumeration: Example:

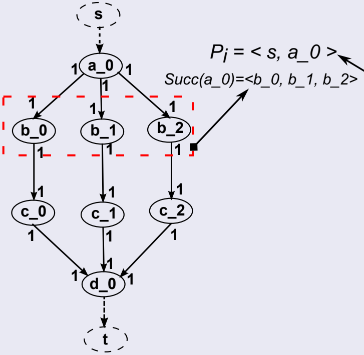


Initial partial path :
$P_i = < s, a\_0 >$

PATHS	Delay
Pi = < s, a_0 >	C_{a_0}

First phase: Finding all possible paths

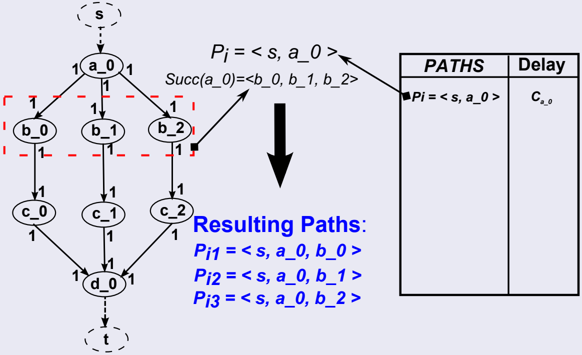
2) Path enumeration: Example:



PATHS	**Delay**
***Pi* = < s, a_0 >**	C_{a_0}

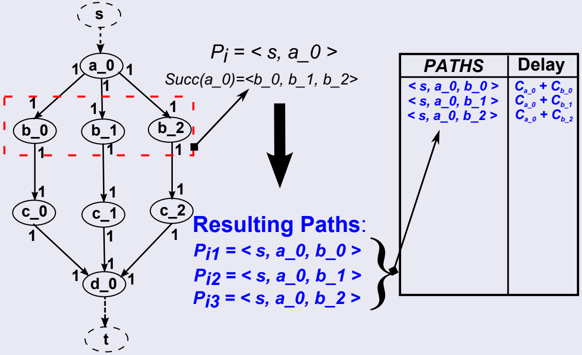
First phase: Finding all possible paths

2) Path enumeration: Example:



First phase: Finding all possible paths

2) Path enumeration: Example:



Second phase: Critical-Path-First (CPF)

Definitions

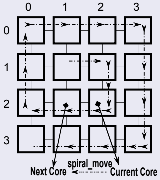
Independent / Dependent Path

A path $P_{A_i} = \langle v_0, v_1, v_2, \ldots, v_j \rangle$ of a certain application A_i is said to be *independent* iff all its actors are unallocated. If at least one of P_{A_i} actors is already allocated, the path is considered *dependent*.

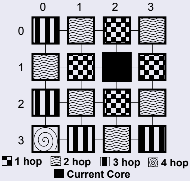
Allocation Condition

$U_{m_i} + u_j \leq 1$

Core Selection



(a) *spiral_move*



(b) *find_nearest_core*

Second phase: Critical-Path-First (CPF)

CPF Algorithm (1/2)

$\boldsymbol{PATHS_{A_i}}$: Lookup table for all possible paths in application A_i ordered according to criticality.

$\boldsymbol{PATHS_G}$: Global lookup table for all $PATHS_{A_i}$ of all applications on the system S.

$\boldsymbol{P_{A_i}}$: A path of application A_i in $PATHS_G$ lookup table, $P_{A_i} = \langle v_0, v_1, v_2, \ldots, v_j \rangle$.

$\boldsymbol{P^p_{A_i}}$: Partial path of full path P_{A_i}

$\boldsymbol{LP^p_{A_i}}$: List of partial paths.

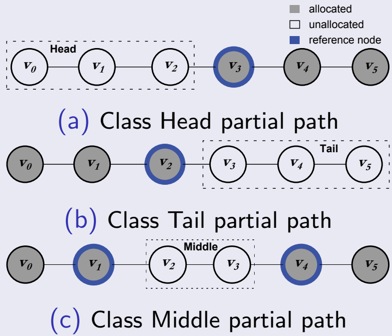
```
begin
    n = spiral_move();
    foreach P_Ai in PATHS_G do
        if P_Ai is Independent then
            foreach v_j in P_Ai do
                while (all cores are not tested) and (v_j not allocated) do
                    if U_mn + u_vj ≤ 1 then
                        allocate v_j on core m_n.
                    else
                        n = spiral_move();
                if v_j not allocated then
                    unallocate ∀v_j ∈ A_i from M.
        else  // Dependent Path Case
            –Dependent Case Next Slide.
```

Second phase: Critical-Path-First (CPF)

CPF Algorithm (2/2)

begin
- n = spiral_move();
- **foreach** P_{A_i} **in** $PATHS_G$ **do**
  - **if** P_{A_i} **is** *Independent* **then**
    - *–Independent Case In Previous Slide.*
  - **else** // Dependent Path Case
    - search for possible $P^p_{A_i}$ in P_{A_i}.
    - classify found $P^p_{A_i}$ & add them to $LP^p_{A_i}$.
    - **foreach** $P^p_{A_i}$ **in** $LP^p_{A_i}$ **do**
      - **if** *Head* **or** *Tail* **then**
        - find the reference actor (Parent).
        - allocate using find_nearest_core.
      - **else if** *Middle* **then**
        - calculate mid-point (core).
        - allocate using find_nearest_core.
      - **if** (v_j **in** $P^p_{A_i}$) *not allocated* **then**
        - unallocate $\forall v_j \in A_i$ from M.

Partial Path Classes



Evaluation Metrics

Two metrics are used to evaluate our approach:

1. Number of allocated applications N.
2. Average end-to-end worst-case response time gain of the applications $R^{av}_{A_{gain}}$.

Also we measured :

- Total utilization of the multi-core processor U_M (the average of all core utilizations, $U_M = \sum_{i=1}^{n} U_{m_i}/n$, where U_{m_i} is the utilization of core i).
- Run-time t_r of the algorithm.

Experimental Setup

- CPF has been evaluated by implementing an allocation tool and experimenting on a set of streaming applications. These streaming applications are taken from the SDF^3 Benchmark.
- The allocation tool instantiates randomized combinations of these applications to create sets of 500 applications.
- Five experiments have been carried out in order to assess the suitability of the proposed approach under different types of applications with different utilizations (High/Low).
- The size of the multi-core platform is an 8x8, 64 core 2D mesh.

Evaluation and Results

Summary of results

$High\%/Low\%$	$100\%/0\%$		$80\%/20\%$		$60\%/40\%$	
Mean of	**CPF**	**FF**	**CPF**	**FF**	**CPF**	**FF**
N	64.1	64.3	98.1	92.1	124.3	117.9
t_r (sec)	2.9	3.1	2.3	2.9	1.8	2.1
$R^{av}_{A_{gain}}$	31.2%		24.4%		22.3%	

$High\%/Low\%$	$40\%/60\%$		$20\%/80\%$	
Mean of	**CPF**	**FF**	**CPF**	**FF**
N	173.6	168.8	300.1	294.9
t_r (sec)	1.3	1.3	0.7	0.4
$R^{av}_{A_{gain}}$	14.1%		8.2%	

Conclusion

- CPF maximizes the overall utilization of the system resources by allocating paths that have the highest impact on the end-to-end response time of the application first.
- CPF is able to minimize the average end-to-end worst-case response time of the applications allocated on the system by enabling application-level parallelism.
- Both algorithms executes in a few seconds, showing that the added complexity is negligible.

The End