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

Hazem Ismail Ali\textsuperscript{1} \quad Luís Miguel Pinho\textsuperscript{1} \quad Benny Akesson\textsuperscript{2}

\textsuperscript{1}CISTER Research Centre/INESC-TEC, Polytechnic Institute of Porto, Portugal

\textsuperscript{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

1. Introduction
   - Background
   - Problem

2. System Model

3. Allocation Algorithm

4. Evaluation and Results
   - Evaluation Metrics
   - Experimental Setup
   - Results

5. Conclusion
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)

- 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.

Figure: An example HSDF graph.
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?
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.
Formally, we consider a system $S$ based on:

**Dataflow (DF)**

$A = \{A_1, A_2, \ldots, A_m\}$

periodic, represented in HSDF. DAGs $G = (V, E)$,

$\zeta = \{\zeta_1, \zeta, \ldots, \zeta_m\}$

Figure: System Model.
Formally, we consider a system $S$ based on:

**Dataflow (DF)**

$A = \{A_1, A_2, \ldots, A_m\}$

periodic, represented in HSDF. DAGs $G = (V, E)$,

$\zeta = \{\zeta_1, \zeta, \ldots, \zeta_m\}$

**Figure**: System Model.
Formally, we consider a system $S$ based on:

- **Dataflow (DF)**
  \[ A = \{A_1, A_2, \ldots, A_m\} \]
  periodic, represented in HSDF, DAGs $G=(V, E)$,
  $\zeta = \{\zeta_1, \zeta_2, \ldots, \zeta_m\}$

- **Multi-core Platform**

- **Core**

- **Interconnected Network (IN)**

- **Allocate**

- **PEDF to schedule allocated tasks**

*Figure*: System Model.
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

\[ \text{Succ}(v_j) = (v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_l}) \]

Resulted Paths:

\[
\begin{align*}
P_{i_1} &= \langle s = v_0, v_1, \ldots, v_j, v_{j_1} \rangle \\
P_{i_2} &= \langle s = v_0, v_1, \ldots, v_j, v_{j_2} \rangle \\
& \vdots \\
P_{i_l} &= \langle s = v_0, v_1, \ldots, v_j, v_{j_l} \rangle 
\end{align*}
\]
First phase: Finding all possible paths

2) Path enumeration: Example:

Initial partial path: $P_i = < s, a_0 >$

PATHS | Delay
--- | ---
$P_i = < s, a_0 >$ | $C_{a_0}$
First phase: Finding all possible paths

2) Path enumeration: Example:

\[ P_i = < s, a_0 > \]

\[ \text{Succ}(a_0) = \langle b_0, b_1, b_2 \rangle \]
First phase: Finding all possible paths

2) Path enumeration: Example:

\[ P_i = \langle s, a_0 \rangle \]

\( \text{Succ}(a_0) = \langle b_0, b_1, b_2 \rangle \)

Resulting Paths:
\[ P_{i1} = \langle s, a_0, b_0 \rangle \]
\[ P_{i2} = \langle s, a_0, b_1 \rangle \]
\[ P_{i3} = \langle s, a_0, b_2 \rangle \]
First phase: Finding all possible paths

2) Path enumeration: Example:

\[ P_j = \langle s, a_0 \rangle \]
\[ \text{Succ}(a_0) = \langle b_0, b_1, b_2 \rangle \]

Resulting Paths:
\[ P_{i1} = \langle s, a_0, b_0 \rangle \]
\[ P_{i2} = \langle s, a_0, b_1 \rangle \]
\[ P_{i3} = \langle s, a_0, b_2 \rangle \]
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

$U_{m_i} + u_j \leq 1$
Second phase: Critical-Path-First (CPF)

**CPF Algorithm (1/2)**

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

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

\( P_{A_i} \): A path of application \( A_i \) in \( PATHS_G \) lookup table,

\( P_{A_i} = \langle v_0, v_1, v_2, \ldots, v_j \rangle \).

\( P^p_{A_i} \): Partial path of full path \( P_{A_i} \).

\( LP^p_{A_i} \): List of partial paths.

begin

\( n = \text{spiral\_move}() \);

foreach \( P_{A_i} \) in \( PATHS_G \) do

if \( P_{A_i} \) is Independent then

foreach \( v_j \) in \( P_{A_i} \) do

while (all cores are not tested) and (\( v_j \) not allocated) do

if \( U_{mn} + u_{vj} \leq 1 \) then

allocate \( v_j \) on core \( m_n \).

else

\( n = \text{spiral\_move}() \).

if \( v_j \) not allocated then

unallocate \( \forall v_j \in A_i \) from \( M \).

else // Dependent Path Case

// Dependent Case Next Slide.

end
Second phase: Critical-Path-First (CPF)

CPF Algorithm (2/2)

begin
n = spiral_move();
foreach PA_i in PATHS_G do
  if PA_i is Independent then
    Independent Case In Previous Slide.
  else // Dependent Path Case
    search for possible P^P_{A_i} in PA_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 ∀v_j ∈ A_i from M.

Partial Path Classes

(a) Class Head partial path
(b) Class Tail partial path
(c) Class Middle partial path
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 $R_{A_{gain}}^{av}$.

Also we measured:

- Total utilization of the multi-core processor $U_M$ (the average of all core utilizations, $U_M = \frac{\sum_{i=1}^{n} U_{m_i}}{n}$, 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\textsuperscript{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

<table>
<thead>
<tr>
<th>High%/Low%</th>
<th>100%/0%</th>
<th>80%/20%</th>
<th>60%/40%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean of</td>
<td>CPF</td>
<td>FF</td>
<td>CPF</td>
</tr>
<tr>
<td>$N$</td>
<td>64.1</td>
<td>64.3</td>
<td>98.1</td>
</tr>
<tr>
<td>$t_r$ (sec)</td>
<td>2.9</td>
<td>3.1</td>
<td>2.3</td>
</tr>
<tr>
<td>$R_{A_{gain}}^{av}$</td>
<td>31.2%</td>
<td>24.4%</td>
<td>22.3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High%/Low%</th>
<th>40%/60%</th>
<th>20%/80%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean of</td>
<td>CPF</td>
<td>FF</td>
</tr>
<tr>
<td>$N$</td>
<td>173.6</td>
<td>168.8</td>
</tr>
<tr>
<td>$t_r$ (sec)</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>$R_{A_{gain}}^{av}$</td>
<td>14.1%</td>
<td>8.2%</td>
</tr>
</tbody>
</table>
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