Exploring Performance Tradeoffs For Clustered VLIW ASIPs

Margarida F. Jacome, Gustavo de Veciana and Viktor Lapinskii
Department of Electrical and Computer Engineering
University of Texas, Austin
Introduction

- Paper proposes an algorithm to support trade-off exploration during the early phases of design/specialization of VLIW ASIPs with clustered datapaths.

- Proposed algorithm explores the space of feasible clustered datapaths and returns:
  - a datapath configuration
  - a binding and scheduling for the operations
  - a corresponding estimate for the best achievable latency over the specified family

- Some experimental results of the proposed algorithm is also provided.
Introduction

- Real time multimedia and signal processing embedded applications spend most of their cycles executing a few time critical code segments (kernels).

- Kernels have well defined characteristics, making them amenable to processor specialization.

- Computation intensive kernels exhibit a high degree of inherent ILP (Instruction Level Parallelism).

- VLIW (Very Large Instruction Word) and ASIPs (Application Specific Instruction Set Processor) provide an attractive solution for such embedded applications.
VLIW Datapaths

- Traditionally, the datapaths of VLIW have been based on a single RF (Register File) shared by all FUs (Functional units).
- The central register provides interconnection among the FUs and to/from memory system. Achieved latency is low.
- As the number of FUs increase; used area, power dissipation and delay grows.

- Key dimension of VLIW ASIP specialization is clustering i.e., the development of datapaths comprised of clusters of FUs connected to the cluster’s register file.
- By using clustering one can reap significant delay, power and area savings.
- On the other hand, Data transfers between RFs may increase latency.
VLIW Datapaths

When considering the specialization of a datapath to a given kernel, one should seek solution with a (large) number of clusters working independently, which decreases power and delay by taking advantage of locality in the computations, while incurring no significant latency/energy penalties due to switching across clusters.
Problem definition

- Kernels are identified and represented as DFGs (DataFlow Graphs).
- DFG model of application may be modified to include nodes corresponding to move/copy operations requiring the interconnect resources.
- Clustered datapaths $D(m,n)$ are parameterized.
  - Datapath may contain several independent components.
    - Component contains collection of clustered FUs
      - Each cluster have no more than $n$ FUs
  - The clusters within each component share a local interconnect structure with capacity not exceeding $m$. 

```
Clustered datapath

interconnect structure
```

$m+1$

$n+1$
Problem definition

- The problem can be stated:
  - The problem $P(m,n,DFG)$ is to find a datapath $D^* \in D(m,n)$, a binding and scheduling of the DFG to $D^*$ that results in a small, if not minimal execution latency. $T^*(m,n,DFG)$ is minimum achievable latency.
Algorithm

Pseudo-code of algorithm:

```
Algorithm(m,n,DFG,TL) {
    TL = max[TL, ASAP(DFG)];  //TL is set to ASAP latency bound for DFG
    solution = (datapath, binding, schedule, latency) = (0,0,0,TL); //initializing solution
    UpdateSolution(solution);
    Set-IDFGs = GenIDFGs(DFG);  //generate a set of IDFGs
    for each IDFG ∈ Set-IDFG {
        s1 = oneClusterSolution(m,n,IDFG);  //try 1 cluster solution
        if (latency(s1) ≤ TL){ UpdateSolution(s1); }
        else {
            s2 = multClusterSolution(m,n,IDFG);  //decomposition 2
            if( latency(s2) < latency(s1) ) {
                UpdateSolution(s2); }
            else { UpdateSolution(s1); }
        }
    }
    return (solution); }
```
Algorithm

- Algorithm goals:
  - Minimize execution latency.
  - Minimize number of data transfers among clusters.

- `UpdateSolution()`
  - Update current global solution, if the sub-problem solution exceeds the current target latency.
  - The sub-problem operations and data transfers are anchored on the corresponding scheduling steps.
  - The mobility of operations not yet anchored is recomputed.

- `latency()`
  - Returns the execution latency of the solution to a sub-problem.
Decomposition

- The algorithm includes two main decompositions:
  - The function **Gen-IDFGs** corresponds to partitioning the DFG into set of IDFGs (independent DFG)
    - IDFGs constitute ideal “chunks” of computation that can be performed on a single datapath component.
      - No need for local computation with other components.
  - IDFG decomposition into the operations, which are the most difficult to handle.
    - Each operation is given a difficulty ranking assessing the likelihood that latency penalty steps will be incurred due to limited cluster or interconnect capacity.

- For each IDFG or IDFG sub-problem the primary goal is to find solution either within the current target latency or resulting in a minimal increase in target latency.
Decomposition and difficulty ranking

- First single cluster solution is generated by calling oneClusterSolution().
- multClusterSolution() generates multiple cluster solution.
  - Solution is decomposed into two sub-problems.
    - First sub-problem is associated with the operations with minimum mobility MM and those with mobility MM+1, if they have a direct producer or consumer with mobility MM. (the most difficult operations)
    - Second sub-problem is associated with operations not considered in the first sub-problem.
  - Sub-problems are decomposed into SCS or MCS
    - MCS is furthermore decomposed into several SCS.
Sub-problem example

IDFG's ASAP schedule  Extracted sub-problem 1

induced sub-IDFG for operations with mobility 0

TL=7

Ranking operations and extracting induced sub problems.
Finding multi-cluster solution for IDFG sub-problems

- There is four steps
  - Vertical Aggregation
    - The idea is to find sets of operations corresponding to sub-trees, rather than the string of the operations.
    - Reduce penalties due to data transfer.
  - Horizontal Aggregation
    - The idea is to identify sets of consecutive operations that have compatible resource requirements.
    - Avoid excessive serialization within limited capacity clusters.
  - Optimization Algorithm
    - To search for a “good” overall partition of sub-IDFG operations and corresponding binding of these partitions to suitable cluster types.
  - Modified List Scheduling
    - Determine execution latencies.
Vertical Aggregation

- Vertigal aggregation of sub-IDFG creates collection of subset of operations \( \mathcal{V} \) corresponding to sub-trees in the sub-IDFG.
- At least two vertical aggregates are required to exploit cluster parallelism.
- Vertigal aggregates are generated in both top-down and bottom-up fashion:
  1. Posit each activity in the top (bottom) layer corresponds to an independent tree.
  2. At each step grow and/or merge trees following the edges between the operations of the new and previous layer only if the resulting aggregates correspond to subtrees and have not resulted in a single aggregate.
  3. If growing/merging of trees violates one of these conditions, restart with the new layer as a new top (bottom) layer.
Vertical Aggregation example
Horizontal Aggregation

- The horizontal aggregation step creates a collection of aggregates $H$ corresponding to sub-IDFG operations on consecutive layers with compatible loads.
  - The load profile of the layer is the sum of the needed ALU and mult operations.
  - Resource load on two layers is compatible if there exists a cluster type that is a good match for both.
  - Resource load of cluster cannot exceed constraint n.
Horizontal Aggregation example

- $n=2$
Optimization algorithm

1. Create coverings of the sub-IDFG’s based on $\mathcal{V}$ and $\mathcal{H}$.
2. Exhaustively derive partitions from each of the obtained covers.
3. Generate alternative multi-cluster datapaths based on partitions.
Optimization algorithm example

- n=2
- m=2 (interconnect capacity)
- Min latency 10

Generating covers, partitions, and deriving clusters types.
Modified list scheduling

- First sub-problem is scheduled using a standard list scheduling priority function, enhanced by tie breaking policy.
  - If operation could not be scheduled within its time frame, TL is incremented, the time frames are updated, and the algorithm is repeated.
- Second sub-problem is scheduled using modified list scheduling algorithm.
  - Algorithm traverses scheduling steps from 1 to TL and, at each step, schedules as many ready operations as available resources permit, using mobility as the priority function.
  - If operation could not be scheduled within its time frame, scheduling operation stops, and a overall scheduling is performed using list scheduling algorithm.
Experimental results

- All data transfers take 1 cycle.
- Interconnect capacity \( m = 2 \)
- Cluster capacity \( n = 2, 3, 4 \)
- \( L \) is latency
- \( DT \) is number of data transfers.
### Experimental results

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>(m,n)</th>
<th>L</th>
<th>Datapath</th>
<th># DTs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DCT-Lee</strong></td>
<td>(2,4)</td>
<td>10</td>
<td>IDFG1: 2(A2M)=2</td>
<td>5</td>
<td>(5+0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: 2(A2M)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,3)</td>
<td>12</td>
<td>IDFG1: (2A1M)+(1A1M)=2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: (2A1M)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,2)</td>
<td>12</td>
<td>IDFG1: 3(A1M)=3</td>
<td>11</td>
<td>(7+4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: 2(A1M)=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DCT-DIF</strong></td>
<td>(2,4)</td>
<td>9</td>
<td>IDFG1: 2(A2M)=2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: (2A1M)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,3)</td>
<td>10</td>
<td>IDFG1: 2(A1M)=2</td>
<td>2</td>
<td>(2+0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: (2A1M)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,2)</td>
<td>13</td>
<td>IDFG1: 2(A1M)=2</td>
<td>2</td>
<td>(2+0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: (2A1M)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DCT-DIT</strong></td>
<td>(2,4)</td>
<td>9</td>
<td>IDFG1: (4A1)+(3A1M)+2(A2M)=4</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: 2(A3A1)=(2A2M)+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,3)</td>
<td>10</td>
<td>IDFG1: 2(A3A1)+(2A1M)=5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDFG2: 2(A2M)+(1A1M)=5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,2)</td>
<td>11</td>
<td>IDFG1: 2(A1M)=7</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

**5th order WDE Filter**: 34 ops (26 add/subs, 8 mults) 1 IDFG, CP=14
- IDFG1: 2(A2M)+(2A1M)=2
- # DTs: 3

**Auto Regression Filter**: 28 ops (12 add/subs, 16 mults) 1 IDFG, CP=8
- IDFG1: 2(A1M)=2
- # DTs: 4

**Avenhaus Filter**: 18 ops (8 add/subs, 10 mults) 1 IDFG, CP=7
- IDFG1: 2(A2M)+(1A1M)=2
- # DTs: 3
Related work

- ASU (Application Specific Unit) based architectural style.
  - ASUs are datapaths whose composition in terms of FUs and interconnection structure is customized to parts of the application flow graph.
  - ASUs do not have permanent storage.
  - Switching of data is done only interconnect and MUXES.
  - Resource sharing is not allowed.

- Retargetable code generation
  - Cluster assignment and other tasks are performed assuming a specific target datapath.