FORMLESS: Scalable Utilization of Embedded Manycores in Streaming Applications

Matin Hashemi, Soheil Ghiasi
Electrical and Computer Engineering Department
University of California, Davis

http://leps.ece.ucdavis.edu

Throughput Estimation:

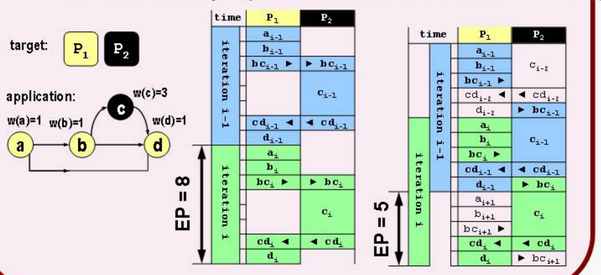
Theorem: Assuming large inter-processor buffers, for any task assignment of dataflow graph G to P processors, there exists an ordering of tasks on processors such that every precedence constraints is met (possibly by overlapping iterations), and the steady state execution period (inverse of throughput) of the application is **EP = MAX workload(p)**.
$1 \le p \le P$

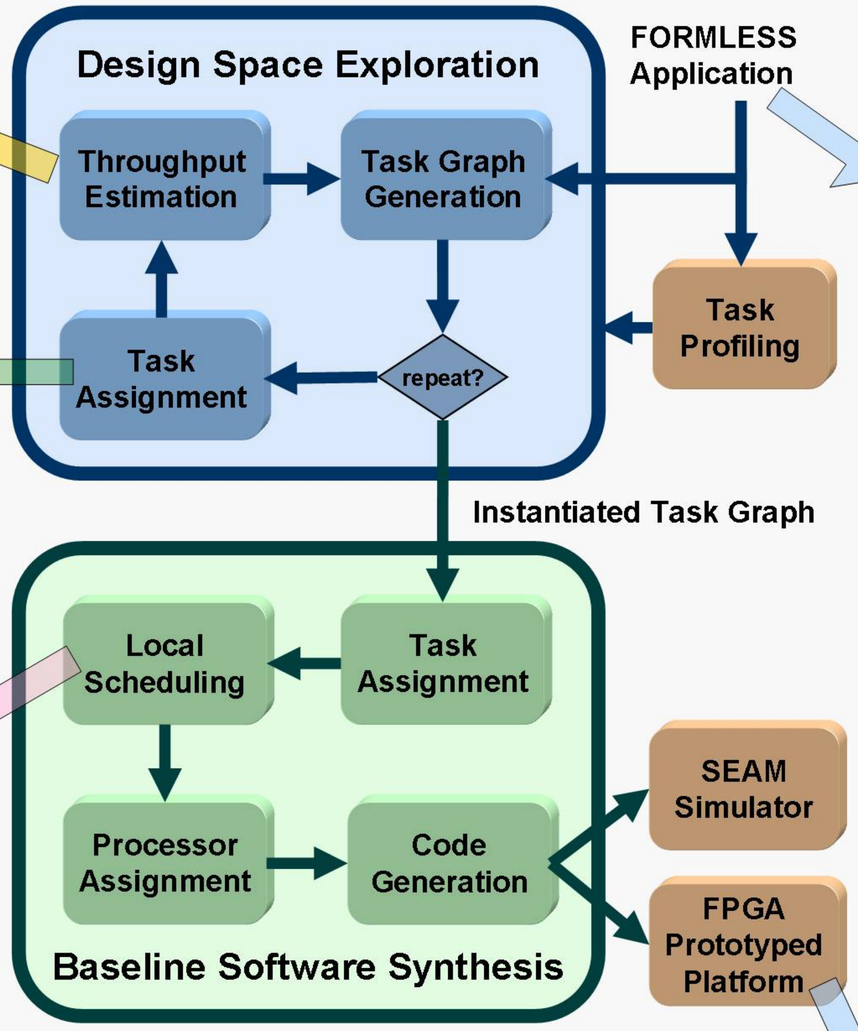
Task Assignment

- Objective: minimize execution period EP
- Constraint: each processor should have at most 4 connections from and 4 connections to other processors
- Implementation:
  - METIS graph partitioning software
  - Post partitioning adjustment to meet the above constraint

Local Scheduling

- Overlap execution of different iterations of the application tasks
  - Mush respect the dependencies
  - Increases throughput
  - Increases memory requirement for inter-task buffers





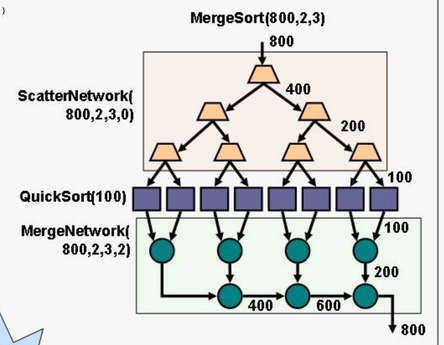

```
//We would like to optimize a merge sort task graph
//which sorts an array of N elements. The rest of the
//parameters should be decided by the DSE tool.
  set K = (2,+1,4); //2..4
  set D = (0,+1,log(N,K)); //0..log(N,K)
  optimize MergeSort ( N, K, D );

//sort an array using a ScatterNetwork to distribute the input
//array among a number of QuickSort actors, and a MergeNetwork
//to merge the sorted sub-arrays into the final output array:
composite_actor MergeSort ( int N, //length of input and output
                            int K, //in-degree of merge actors
                            int D  //depth of the tree
                          ){
  interface {
    input input_array ( N );
    output output_array ( N );
  }
  composition {
    if (D==0) {
      instantiate QuickSort quicksort ( N );
      connect ( input_array,  quicksort.unsorted_array );
      connect ( quicksort.sorted_array,   output_array );
    } else {
      //instantiate a scatter network which distributes
      //the input array among K^D sort actors
      instantiate ScatterNetwork scatternet( N, K, D, explore);

      //instantiate the sort actors
      for (i=0; i < K^D; i++)
        instantiate QuickSort quicksort[i] ( N / K^D );

      //instantiate a merge network which merges
      //the sorted sub-arrays into the output array
      instantiate MergeNetwork mergenet ( N, K, D, explore );

      //connections:
      connect ( input_array, scatternet.input_array );
      for (i=0; i < K^D; i++) {
        connect ( scatternet.sub_array[i],
                  quicksort[i].unsorted_array );
        connect ( quicksort[i].sorted_array,
                  mergenet.sub_array[i] );
      }
      connect ( mergenet.merged_array, output_array );
    }
  }
}
```




```
//sort an array:
actor QuickSort ( int N //length of input and output arrays
                ){
  interface {
    input  unsorted_array ( N );
    output sorted_array ( N );
  }
  function {
    //the quicksort algorithm
  }
}
```

QuickSort(100) 100 100

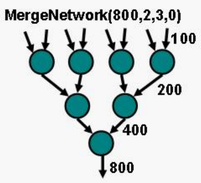
```
//merge input sorted arrays into one larger output array:
actor Merge ( int   N, //length of the output array
              int   K, //number of the input arrays (in-degree)
              int[] M  //length of the input arrays
            ){
  interface {
    output merged_array ( N );
    for ( i=0; i < K; i++ ) {
      if (M != null)  m = M[i];
      else            m = N/K;
      input sub_array[i] ( m );
    }
  }
  function {
    //merge based on the method in mergesort algorithm
  }
}
```

Merge(600,3,null) 200 200 200 600

```
//merge input sorted arrays into one larger output array
//using a tree network constructed out of merge actors:
composite_actor MergeNetwork ( int N, //length of the output array
                               int K, //in-degree of merge actors in the tree
                               int D, //depth of the tree
                               int L  //the first L levels of the tree are
                                      //constructed in a linear structure.
                             ){
  explore(
    set L = (0,+1,D);
  )
  interface {
    output merged_array ( N );
    for (i=0; i < K^D; i++)
      input sub_array[i] ( N / K^D );
  }
  composition {
    //instantiate merge actors in tree structure (levels L to D-1)
    for (d=D-1; d >= L; d--)
    for (i=0; i < K^d; i++)
      instantiate Merge merge[d][i] ( N / K^d, K, null );

    //instantiate merge actors in linear structure
    int S = N / K^L;
    for (i=1; i < K^L; i++)
      instantiate Merge merge[0][i] ( S*(i+1), 2, (S*i, S) );

    //connections:
    //input
    for (i=0; i < K^D; i++)
      connect ( sub_array[i], merge[D-1][i/K].sub_array[i%K] );
    //tree structure
    for (d=D-1; d > L; d--)
    for (i=0; i < K^d; i++)
      connect ( merge[d][i].merged_array, merge[d-1][i/K].sub_array[i%K] );
    //linear structure
    for (i=1; i < K^L; i++) {
      if (i==1) connect ( merge[L][0].merged_array,   merge[0][i].sub_array[0] );
      else      connect ( merge[0][i-1].merged_array, merge[0][i].sub_array[0] );
      connect ( merge[L][i].merged_array, merge[0][i].sub_array[1] );
    }
    //output
    connect ( merge[0][K^L-1].merged_array, merged_array );
  }
}
```




```
composite_actor ScatterNetwork //details removed for brevity
```

FPGA Prototyped Platform

- Altera DE4 FPGA board
- 16 NiosII/f cores
- instruction cache: 8KB
- data cache: 32KB
- FIFO depth: 1024