Locality management using multiple SPMs on the Multi-Level Computing Architecture

Ahmed M. Abdelkhalek and Tarek S. Abdelrahman

The Edward S. Rogers Department of Electrical and Computer Engineering
University of Toronto

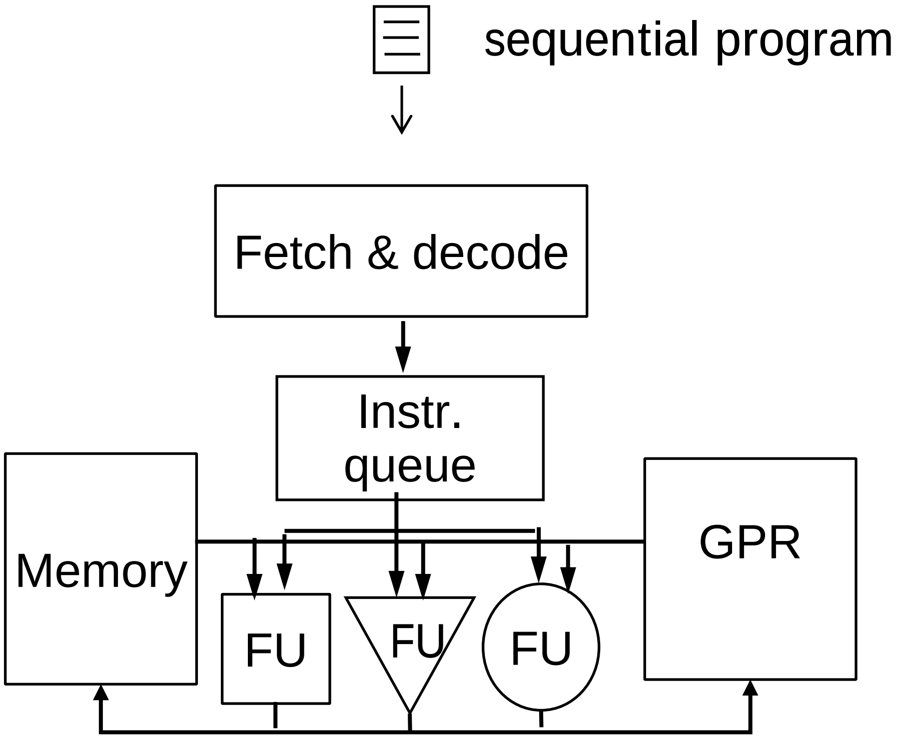
Oct. 26th, 2006
ESTIMedia, Seoul, Korea

Motivation for MLCA

1. Parallel programming is **difficult**
2. Need flexible MP-SoC architectures

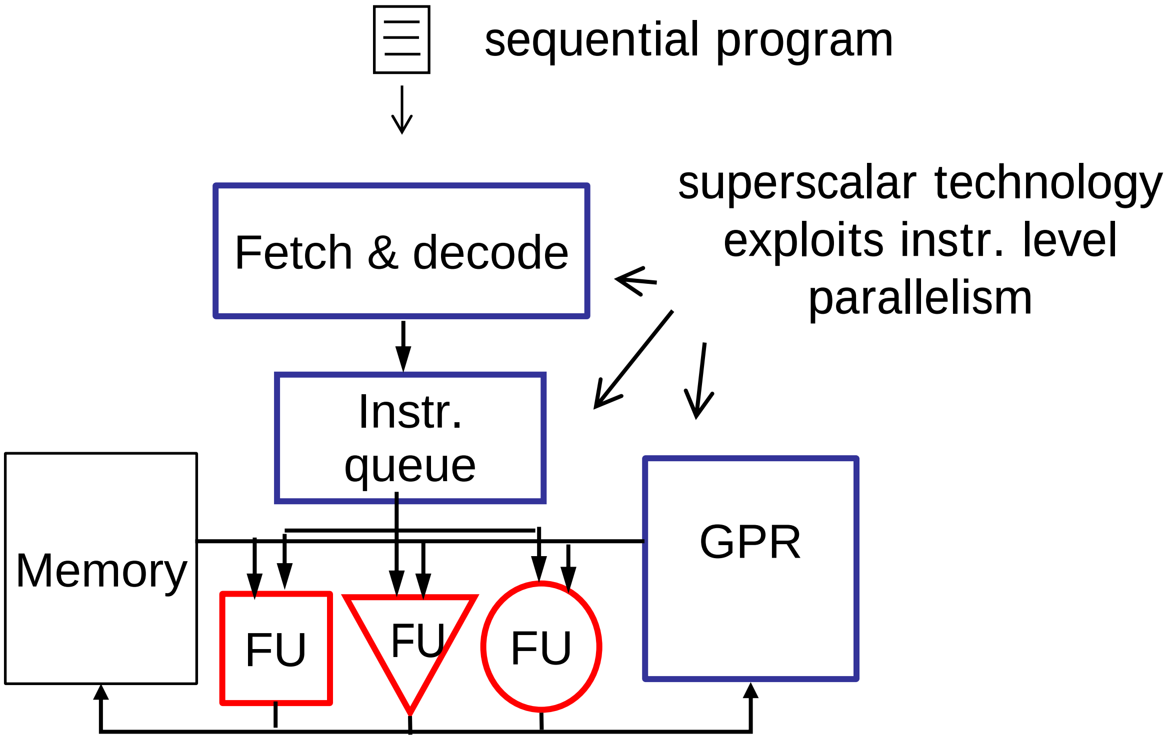
- Developed by:
  - F. Karim, A. Mellan, A. Nguyen - *STMicroelectronics*
  - U. Aydonat, T. Abdelrahman - *Univ. of Toronto*
  - "A Multi-Level Computing Architecture for Multimedia Applications"
    - *IEEE Micro, vol. 24, no. 3, 2004*

What is the MLCA?



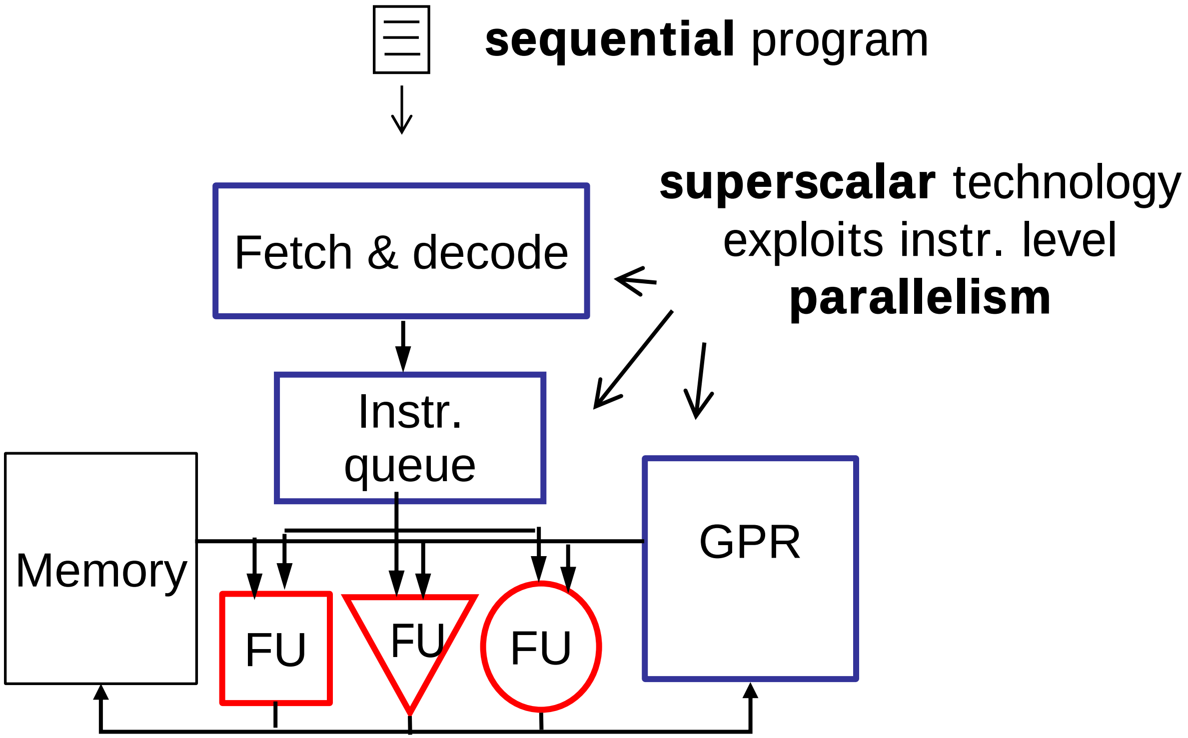
Abstract micro-processor architecture

What is the MLCA?



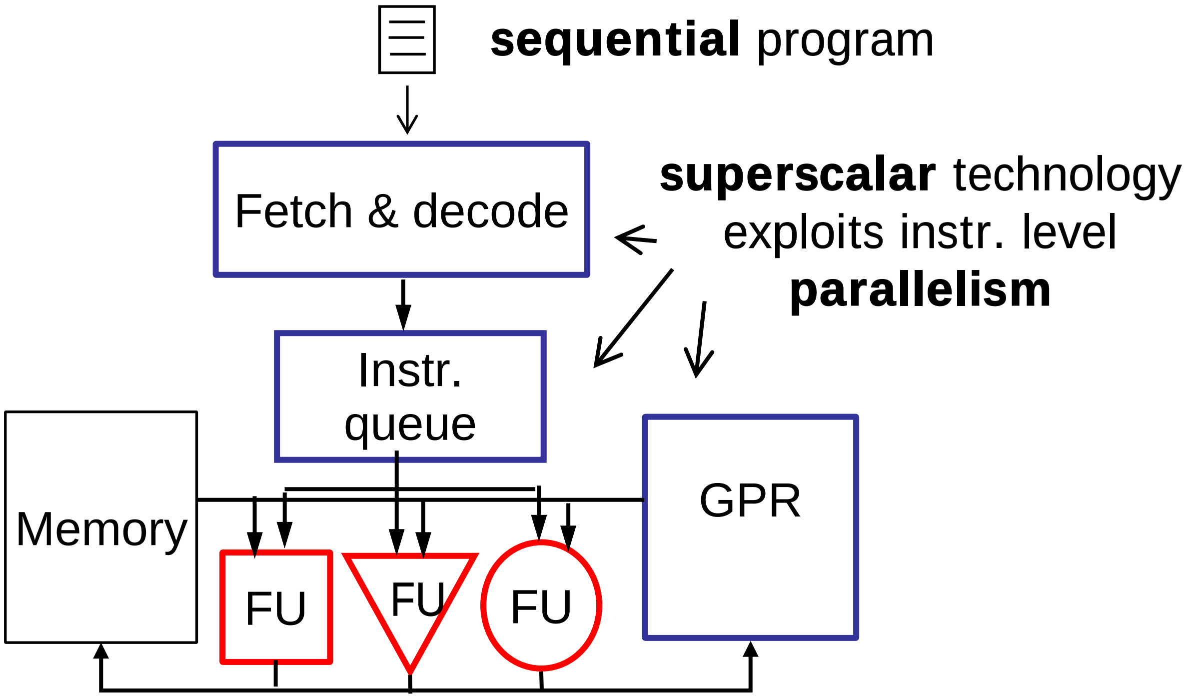
Abstract micro-processor architecture

What is the MLCA?



Abstract micro-processor architecture

What is the MLCA?



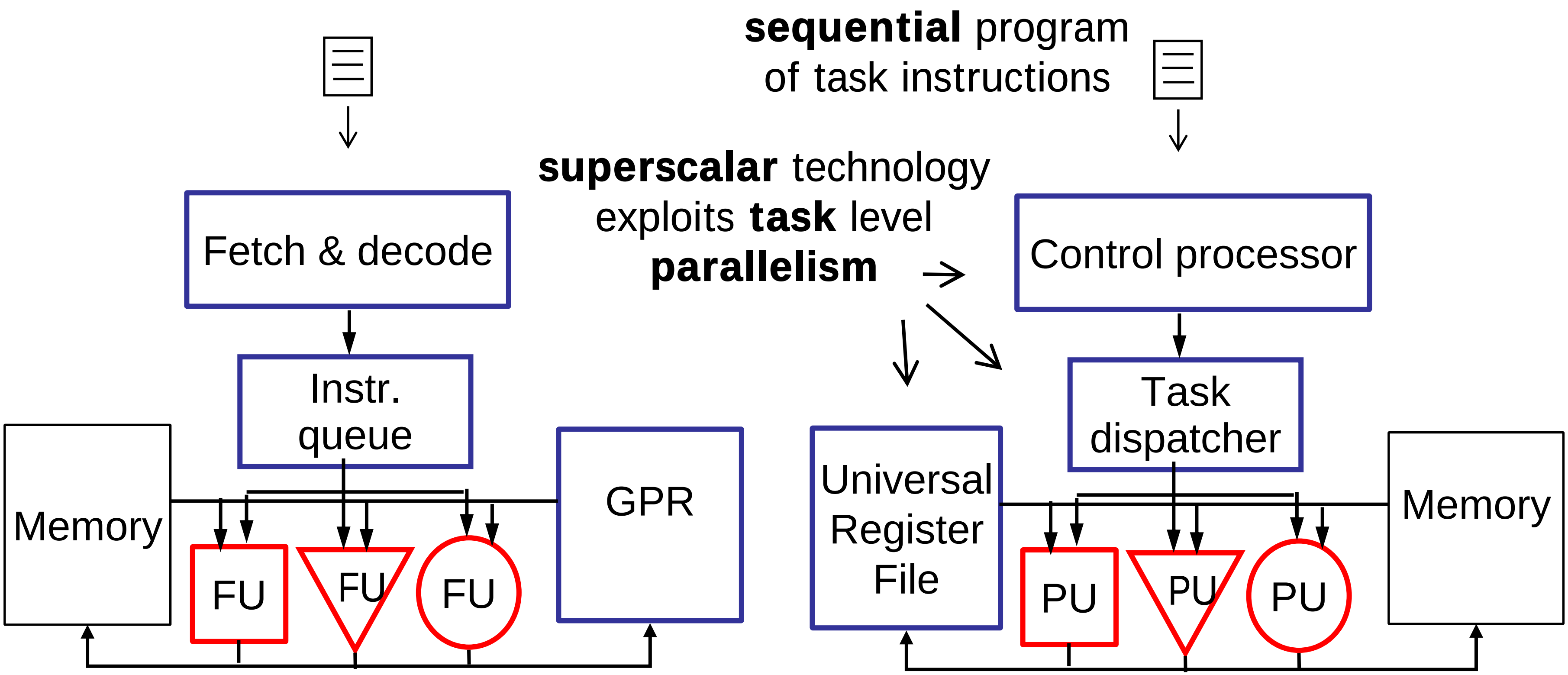
sequential program

superscalar technology exploits instr. level **parallelism**

Abstract micro-processor architecture

Isn’t parallel execution the goal of parallel programming?

What is the MLCA?



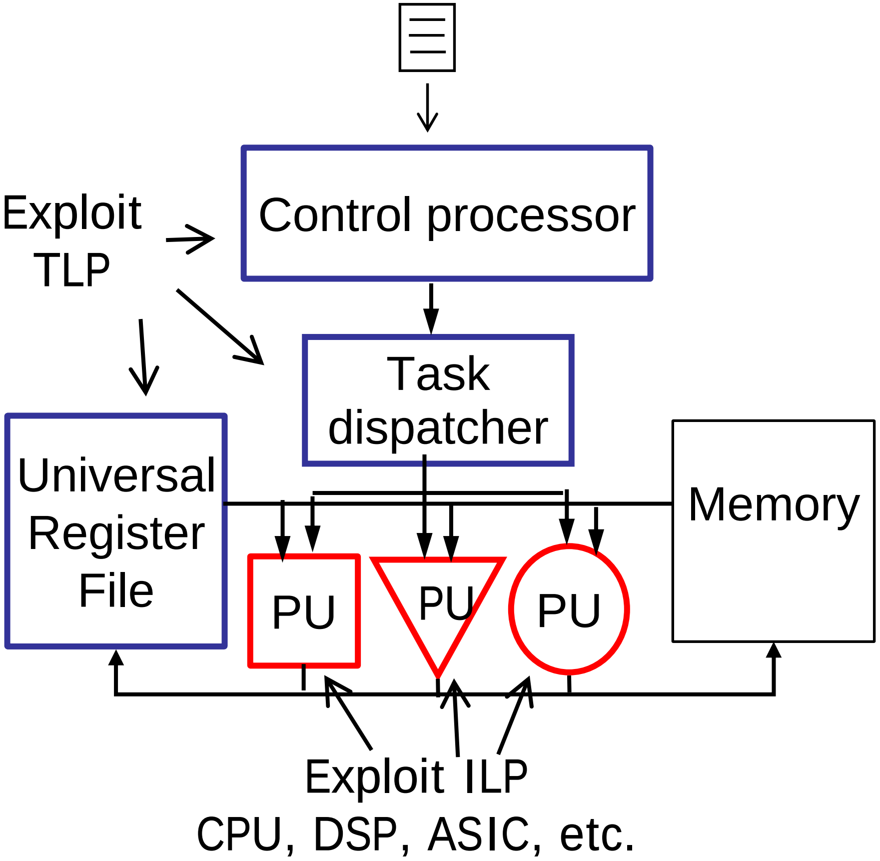
Abstract micro-processor architecture

Abstract
Multi-Level Computing Architecture

What is the Multi-Level Computing Architecture?

- Novel flexible MP-SoC architecture
- New parallel programming model
  - Targets application TLP and ILP
- Uses layered approach in HW and SW
  - Upper layer exploits TLP
    - HW: control processor, task dispatcher, and universal register file (URF)
    - SW: **control** program
  - Lower layer exploits ILP
    - HW: processing units
    - SW: task functions

MLCA Architecture & Programming Model



Sample control program

```
do {
    notzero = Add (in v1, in v2, out v3);
    if (notzero)
            Div (in v3, in v4, out v5);
    done = CheckDone (in v4, in v6, out v3);
} while (done==0);
```

Sample task function

```
int Add () {
    int n1 = readArg(0);
    int n2 = readArg(1);
    writeArg(0, n1+n2);
    return (n1+n2)!=0;
}
```

MLCA Architecture & Programming Model

- Reduced SW complexity:
  - no explicit parallel programming
  - synchronization and communication separate from actual computations
- Automatic extraction of parallelism
  - superscalar technology
- Flexibility
  - PU number/types
  - memory hierarchy
  - scheduling policy

Sample control program

```
do {
    notzero = Add (in v1, in v2, out v3);
    if (notzero)
            Div (in v3, in v4, out v5);
    done = CheckDone (in v4, in v6, out v3);
} while (done==0);
```

Sample task function

```
int Add () {
    int n1 = readArg(0);
    int n2 = readArg(1);
    writeArg(0, n1+n2);
    return (n1+n2)!=0;
}
```

MLCA Architecture & Programming Model

- Optimizing system
  - How divide application into tasks?
  - How decide on task arguments?
  - Application-architecture matching
- Simple path to initial solution exists

Sample control program

```
do {
    notzero = Add (in v1, in v2, out v3);
    if (notzero)
            Div (in v3, in v4, out v5);
    done = CheckDone (in v4, in v6, out v3);
} while (done==0);
```

Sample task function

```
int Add () {
    int n1 = readArg(0);
    int n2 = readArg(1);
    writeArg(0, n1+n2);
    return (n1+n2)!=0;
}
```

Outline

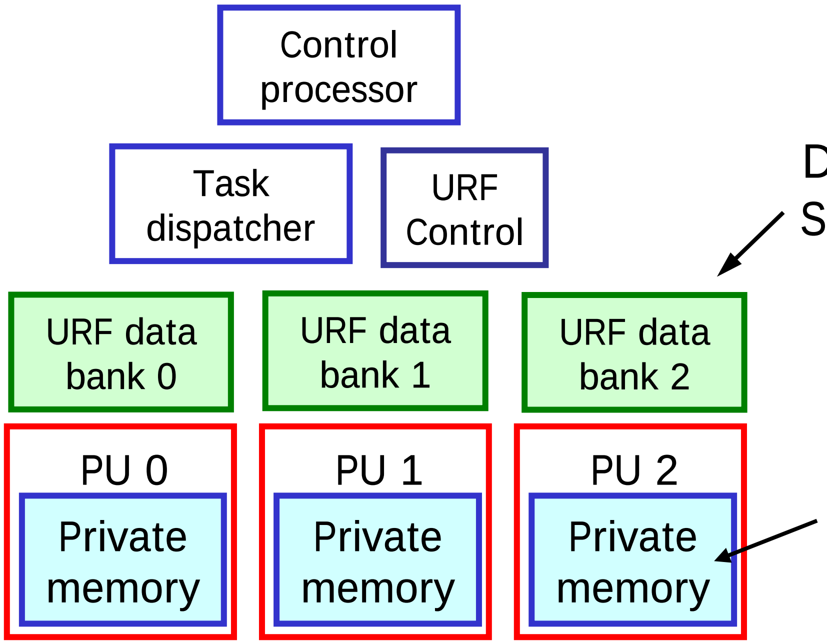
- MLCA intro
- Motivation
- Target MLCA
- Problem definition
- Global task data mgmt
- Evaluation
- Conclusion

Motivation

- MLCA *flexible* architecture:
  - Opportunity for optimization
  - Focus on memory hierarchy
- Silicon technology scaling:
  - Performance improving faster for gates than wires
  - Cross-chip communication becoming more expensive
- Avoid centralized memory:
  - Better scalability for future MLCA chips

Target MLCA

- MLCA naturally breaks down data into two types:
  - **Intra**-task data: created and destroyed by task each time it executes, not needed by other tasks
  - **Inter**-task data: needed by more than one task, identified through the URF



Distributed-shared (NUMA) Scratch-Pad Memory banks store **inter**-task data

store **intra**-task data

Problem definition

- How do we *efficiently* use the target MLCA?
  - How is global data allocated in the distributed banks?
  - How to ensure access locality?
  - Use static approach or allow dynamic data movement between banks?
- How to easily integrate with MLCA 2-level programming model?
- Focus on global data mgmt only
  - Local task data handled by PU cache, etc.

Goal: better performance and easy-to-use

Global task data mgmt

- Approach:
  - Minimize cross-chip communication
  - Execute task on PU near bank with global data it needs
- Methodology:
  - **Bank memory allocation**: task creates data in certain bank
  - **Task-bank association**: indicate preference of where to schedule
  - **Bank data replication/migration**: copy/move global data between banks
  - Appropriate task scheduling policies
  - Easy to use in control program

Example control program

Bank identifier

```
while (...) {
      setup (out x bank 1,
                 out y bank 2,                          // bank memory allocation
                 out z bank 3);

      taskA (in x, out x) on bank 1;                    // task-bank association
      taskB (in y) on bank 2;
      taskC (in z) on bank 3;

      move  x, bank 3;                                  // bank data migration
      copy   y, ycopy, bank 3;                          // bank data replication
      taskD  (in x, in ycopy, in z) on bank 3;
      ...
}
```

- Problem with loops:
  - All iterations use same sets of banks
  - Not desirable with independent iterations

Example control program

Virtual bank number

```
while (...) {
    setup (out x bank 1, out y bank 2, out z bank 3);  // bank memory allocation

    taskA (in x, out x) on bank 1;                // task-bank association
    taskB (in y) on bank 2;
    taskC (in z) on bank 3;

    move  x, bank 3;                              // bank data migration
    copy  y, ycopy, bank 3;                       // bank data replication
    taskD  (in x, in ycopy, in z) on bank 3;
    ...
    remap bank 1, bank 2, bank 3;                 // bank remapping
}
```

- Solution for loops:
  - Application uses virtual bank numbers
  - Virtual numbers mapped to physical ones at run-time
  - **Bank remapping**: indicate next iteration can use different banks

Example control program

Virtual bank number

```
while (...) {
      setup (out x bank 1, out y bank 2, out z bank 3);  // bank memory allocation

      taskA (in x, out x) on bank 1;                     // task-bank association
      taskB (in y) on bank 2;
      taskC (in z) on bank 3;

      move  x, bank 3;                                   // bank data migration
      copy   y, ycopy, bank 3;                           // bank data replication
      taskD  (in x, in ycopy, in z) on bank 3;
      ...
      remap bank 1, bank 2, bank 3;                      // bank remapping
}
```

- Focus on optimization not correctness
  - Limit copies to constant data

Task scheduling policies

- Task-bank association serves as *hint* to scheduler
- Various ways to deal with at run-time:
  - Completely ignore
    - E.g. schedule first ready task on any PU
  - Strictly adhere to
    - E.g. only schedule task on PU preference
  - Somewhere in between
    - E.g. schedule on preference, but ignore if wait too long

Evaluation

- MLCA simulator
  - C++/SystemC timed functional simulator
- Media applications:
  - MP3 decoder, FM radio demodulator, GSM voice encoder
- Evaluate against:
  - minimum support needed to use target MLCA
  - round-robin for data allocation in banks and for task scheduling
- Vary NUMA-ness of bank accesses

Results

- Impact of individual techniques:
  - Bank memory allocation and task-bank association: up to 21%
  - Bank remapping: up to 18%
  - Bank data replication/migration: up to 22%
- Applications with various types of parallelism can benefit:
  - GSM: pipeline //ism across iterations: 19%
  - MP3: //ism within iteration and coarse pipeline //ism: 33%
  - FMR: //ism within iteration and fine pipeline //ism: 40%
- Impact increases with NUMA-ness of banks
  - All apps benefit when remote bank access >= 14 PU cycles
- Scheduling policies that favor local access are necessary

Only 6-14% potential for improvement remaining!

Conclusion

- This work:
  - Introduced distributed-shared memory MLCA
  - Solution for global task data mgmt
    - programming directives
    - task scheduling policies
  - Showed effectiveness of our approach at improving performance
- Future work:
  - Compiler support
  - Hardware evaluation

Thank you!

Questions / Comments?

Task scheduling policies

- FR:
  - first ready task on first ready PU, visit PUs in round-robin fashion
- CL:
  - first ready task on closest ready PU
- POLA:
  - schedule task only on PU near bank preference, allow look ahead down task ready queue
- POLATO:
  - same as POLA but timeout after certain threshold and revert back to FR
- POLAEM/POLAEMTO:
  - same as POLA/POLATO but apply bank preference scheduling on moves and copies as well