Ptolemy Miniconference, March 1995

Parallel Implementation Techniques for Embedded DSP Systems

S. Sriram

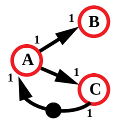
Prof. Edward A. Lee

Parallelism in Embedded DSP Systems

- **Parallelism : concurrency at the system level**
  - **target a system consisting of a mix of dedicated parts such as FFT chips and programmable DSPs**
- **Embedded : low cost, dedicated multiprocessor**
  - **Examples — multimedia: set-top boxes, multimedia workstations, communications: digital cell phones**
- **Motivation**
  - **high throughput applications demand processing power**
  - **use of commodity programmable parts: attractive alternative to ASICs**
  - **advantages of software solutions**
  - **silicon technology: multi-DSP chips available from number of companies**

Parallel Implementation of DSP Algorithms Static

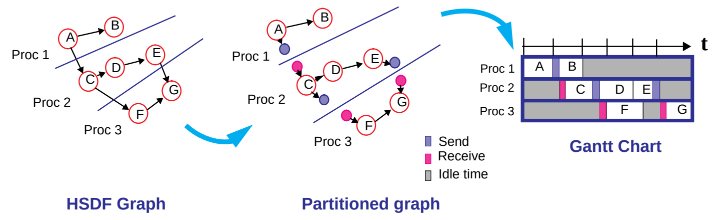
- **Issues: scheduling, interprocessor communication (IPC) & synchronization overhead, hard real-time requirement**
  - **computation on unbounded data streams**
- **Our strategy**
  - **Restricted application domain: Synchronous Dataflow and extensions**



  - **Well-defined methodology:**
    - Compilation from dataflow graphs
    - Extensive use of compile-time scheduling techniques
  - **Given this methodology optimize hardware architecture and parallel implementation:**
    - Reduce IPC overhead: Ordered Transactions scheme
    - Reduce synchronization overhead

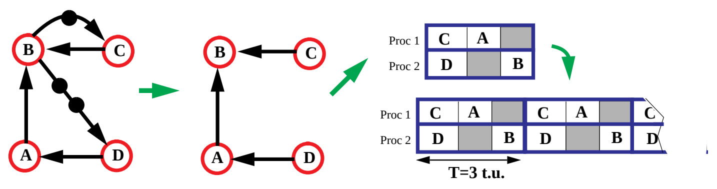
Scheduling

- **Scheduling homogeneous SDF graphs**
  - **Assigning actors to processor: Assignment**
  - **Determining the order of execution of actors on a processor: Ordering**
  - **Determining when an actor actually fires: Firing times**
- **Dynamic (run time) vs. Static (compile time) strategies**
- **Use execution time estimates**
  - **Fully Static : all three scheduling steps performed at compile time, assuming execution times estimates are precise**



Blocked Multiprocessor Schedules

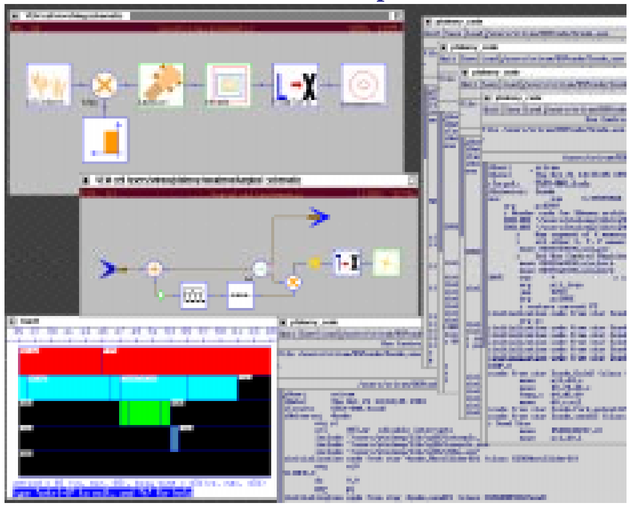
- Homogeneous SDF graph converted to an Acyclic Precedence Graph (APG) by removing edges with delays
  - Intra-iteration precedences ignored during scheduling



- Minimize *T*: classical MP scheduling from an APG
  - Optimal scheduling under resource constraints is intractable (NP-Hard)
  - Several heuristics exist: list scheduling [Hu 61], [Sih 92], [Sarkar 89], ...
  - Unfolding (increased blocking factor) and retiming transformations

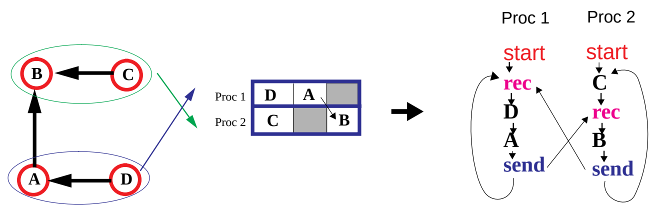
Implementation in Ptolemy

Block Diagram ➜ SDF Graph ➜ Homogeneous APG ➜ Parallel Schedule ➜ Multiprocessor code



Self-timed Scheduling

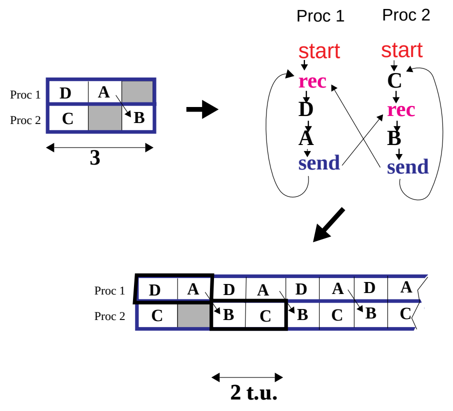
- Fully-static schedule assumes knowledge of exact actor execution times - not always practical:
  - compilation from high-level language, error handling, unpredictable execution times due to instruction-level parallelism
- Model followed in Ptolemy
  - reasonably good estimates of execution times known at compile time
  - construct fully-static schedule, ignore exact timing information



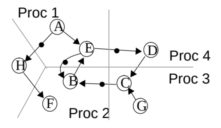
- Larger run time overhead compared to fully-static sched.

Communication Pattern

- Attempt to predict run time inter-processor communication pattern and use this information to optimize parallel implementation



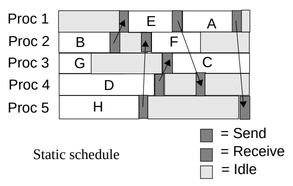
Communication Pattern



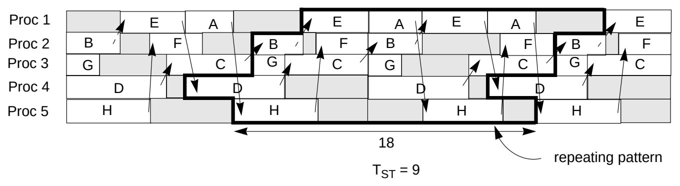
DFG "G"

Execution Times

A , B, F	: 3
C, H	: 5
D	: 6
E	: 4
G	: 2

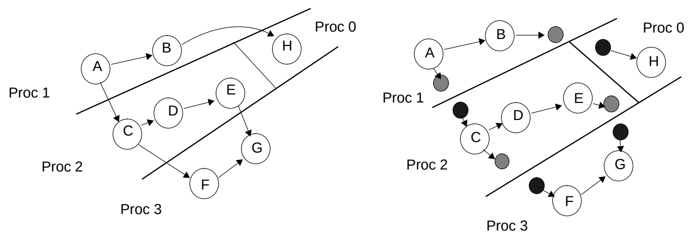


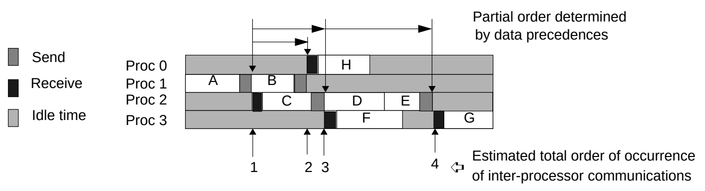
Static schedule



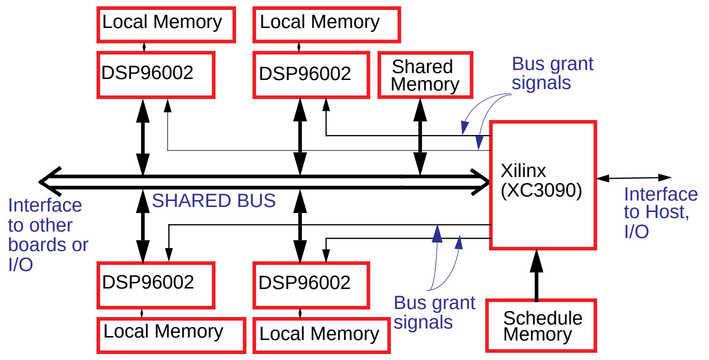
$T_{ST} = 9$

Ordered Transactions





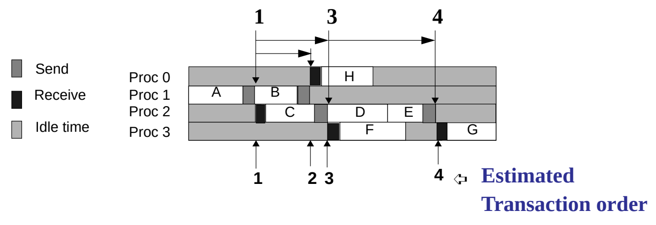
Ordered Memory Access Architecture



- Low overhead IPC (3 instruction cycles)
- No need for explicit synchronization
- Performance degrades if execution times vary at run time (or if compile estimates are bad), computations are correct

Choosing a Transaction Order

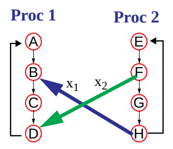
- Transaction order imposes run time constraints absent in the unconstrained self-timed schedule
  - Partial order due to precedence constraints: 2 » 1, 4 » 3 » 1
  - Any total order is a valid transaction order



- Naive transaction order derived from one block of the schedule: not always the best choice
  - Can efficiently determine a transaction order that is "optimal"

Minimizing Synchronization in self-timed schedules

- Self-timed scheduling: each inter-processor communication point is also a synchronization point
  - sender needs to check for buffer overflow
  - receiver needs to check for buffer empty
- Compile time analysis of schedule can reduce this overhead
  - Sender synchronizations are eliminated by sizing buffers appropriately
  - Remove redundant receiver synchronizations: synch x2 is redundant



  - Perform transformations on the schedule: introduce new synchronization points

Conclusions

- Discussed mechanism for constructing parallel schedules from SDF graphs
- Discussed how compile time scheduling can be effectively employed for SDF applications
- Discussed parallel code generation methodology in Ptolemy
- Presented the ordered transactions approach: hardware architecture optimized for the self-timed strategy employed in Ptolemy
- Described minimization of synchronization costs by means of compile-time analysis