Flextream: Adaptive Compilation of Streaming Applications for Heterogenous Architectures

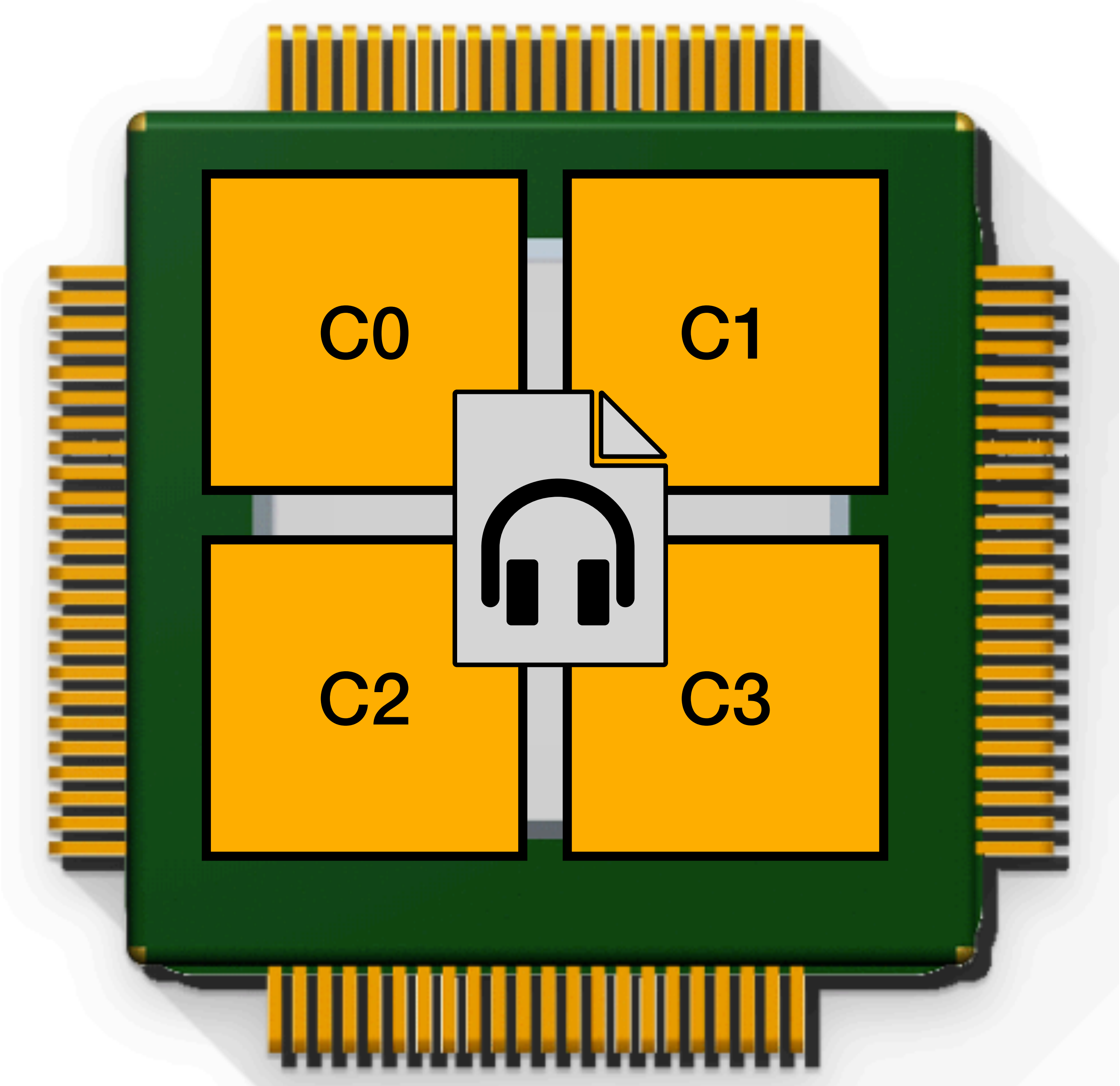
Joshua Dela Rosa
2024324064

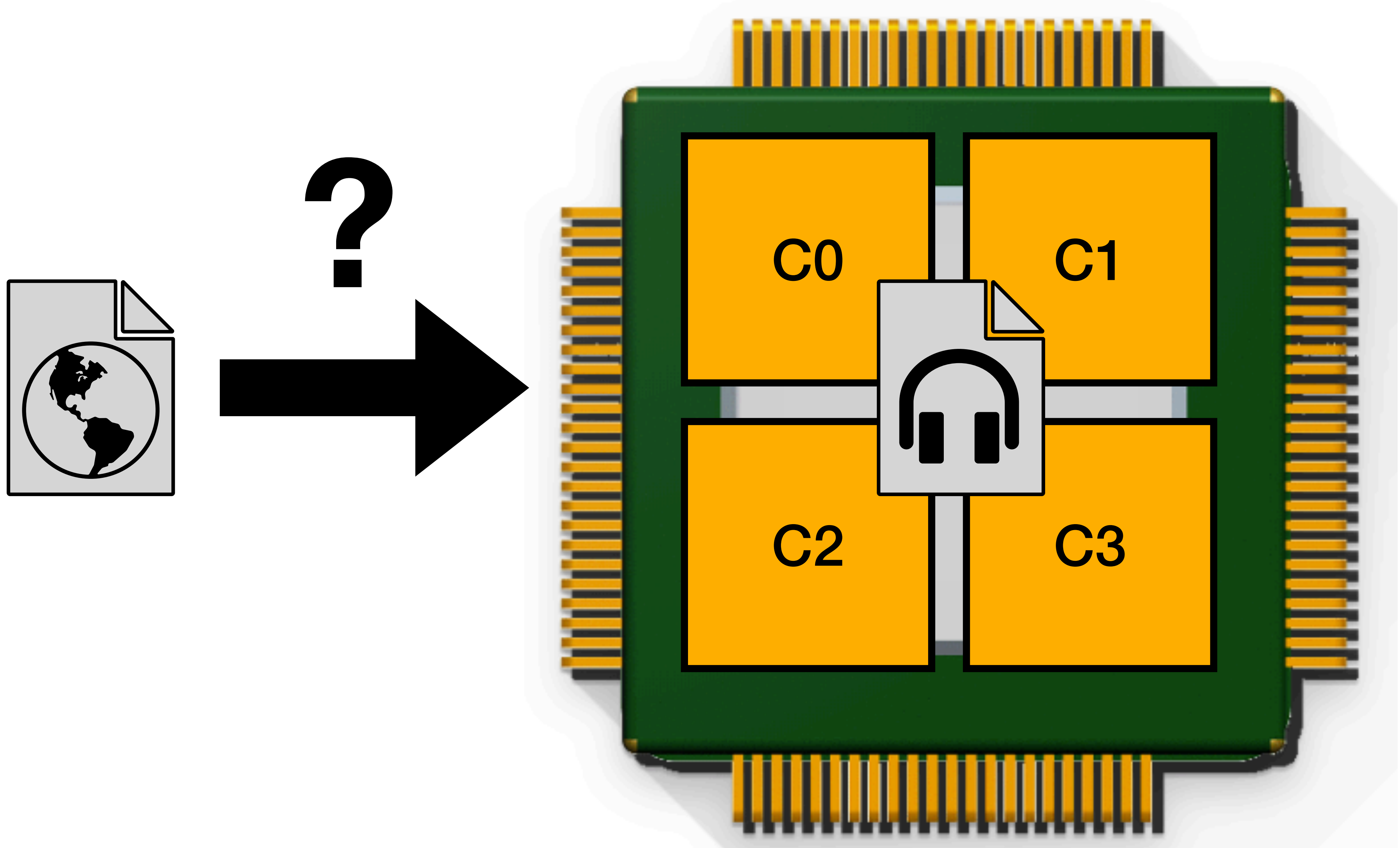
2025-10-13

Introduction

- Performance & Efficiency Demand $\Longrightarrow$ Multicore systems
- Biggest challenge is the need for applications to adapt to dynamic changes in resources.
- Need strategy to effectively allocate resources dynamically.

Example





On its own, music player can exploit all resources.

If the user starts browsing the web, available resources must adjust.

Compile Approaches

Static Compilation

- Can generate high quality resource allocations on the spot.
- Sensitive to changes in resources.
- Compile multiple versions of the app?
  - Code bloat

Dynamic Compilation

- Compile the app repeatedly on runtime.
- Adapts when resources changes.
- Only promising if cost of compilation is low.

Flextream

Static Compilation

- Can gen
  allocatio
- Sensitive
- Compile
  app?
  - Code bloat

Dynamic Compilation

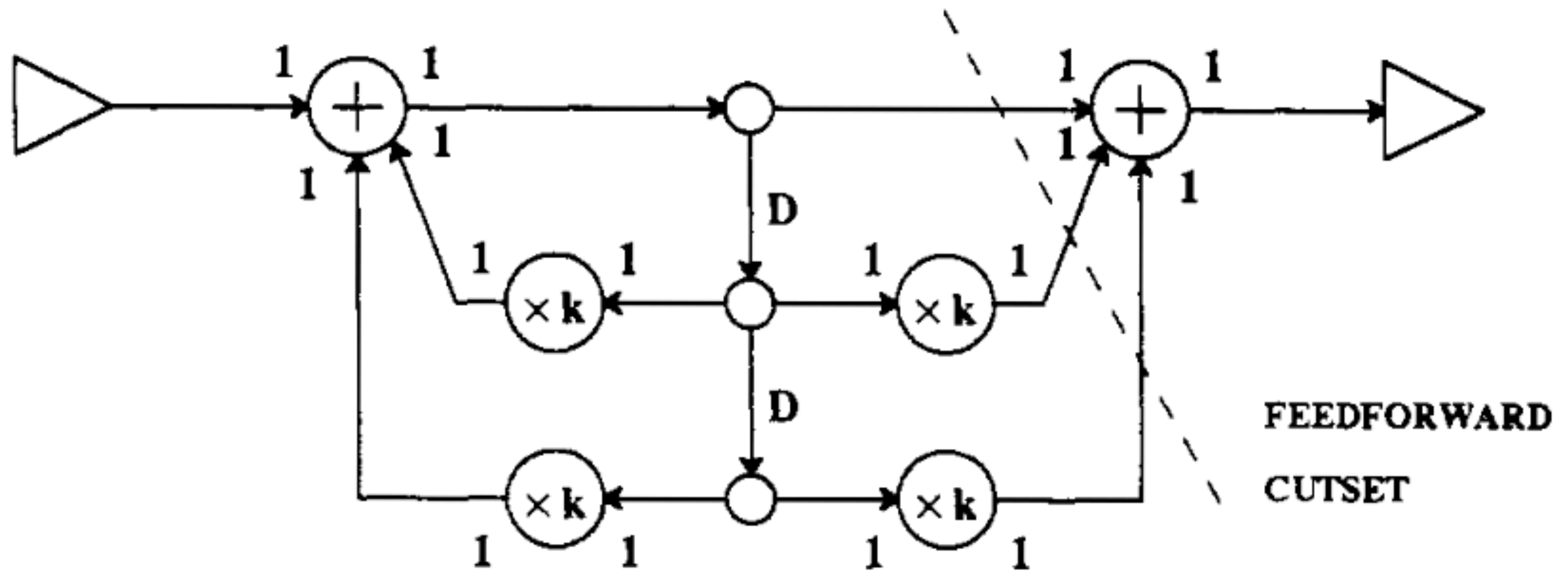
- edly on
- changes.
- f compilation

FLEXTREAM

- Compilation and runtime adaptation system.
- Combines the benefits of static scheduling and dynamic adaptation.
- Target *streaming* applications.

Streaming Application

- A program represented as a graph,
- Each node is a computation,
- Edges between nodes describe dataflow.
- A node is called "actor" in this paper.



- A stream program (graph) is mapped to many-core heterogenous architecture by assigning nodes to cores.

Image Source: E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in Proceedings of the IEEE, vol. 75, no. 9, pp. 1235-1245, Sept. 1987, doi: 10.1109/PROC.1987.13876.

Flextream

Compiler Framework

- Objective is to obtain a schedule that produces the maximum throughput of the stream graph,
    - while considering computation/communication overheads, and memory requirements.

- Compilation divided into two phases,
    1. static compilation
    2. online (dynamic) adaptation.

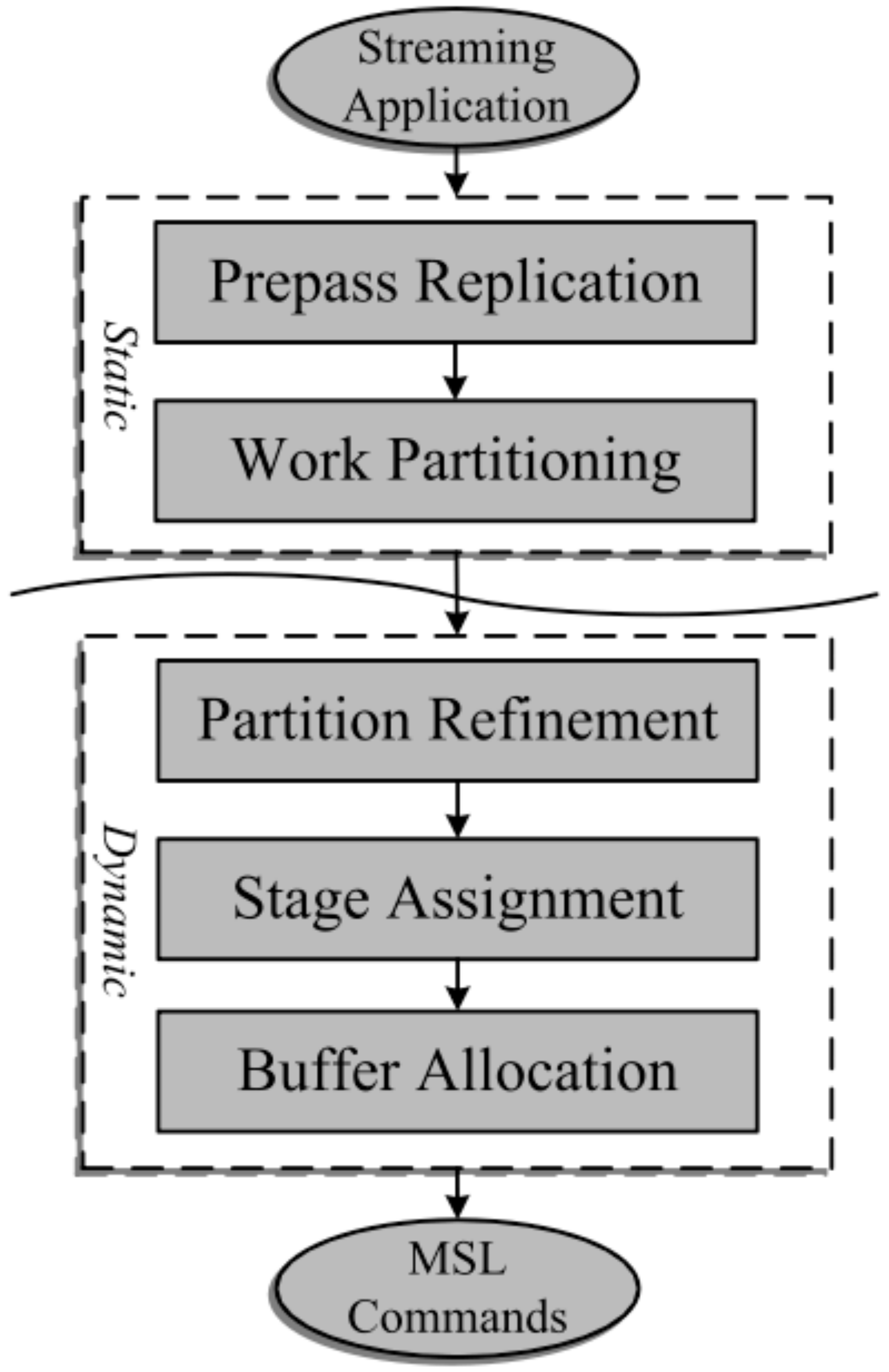


Figure 2: General flow of the Flextream framework

Flextream Intuition

- Statically compile a program assuming all resources are available.
- Using the solution (optimal schedule) found, dynamically adapt to the actual available resources in the system.
- At runtime, every time the resources change, adapt.

- By starting at an optimal solution, the schedule can be adjusted to a system with less resources but close enough to the performance of an optimal solution.

Static Compilation

Static Compilation

- Goal: find an optimal schedule assuming all resources are available for the app.
- Starting from ideal case allows for more flexibility when adapting to non-ideal case.
- Steps:
  1. **Prepass replication**: to adjust the amount of parallelism
  2. **Work partitioning**: map actors to processors using integer linear programming (ILP)

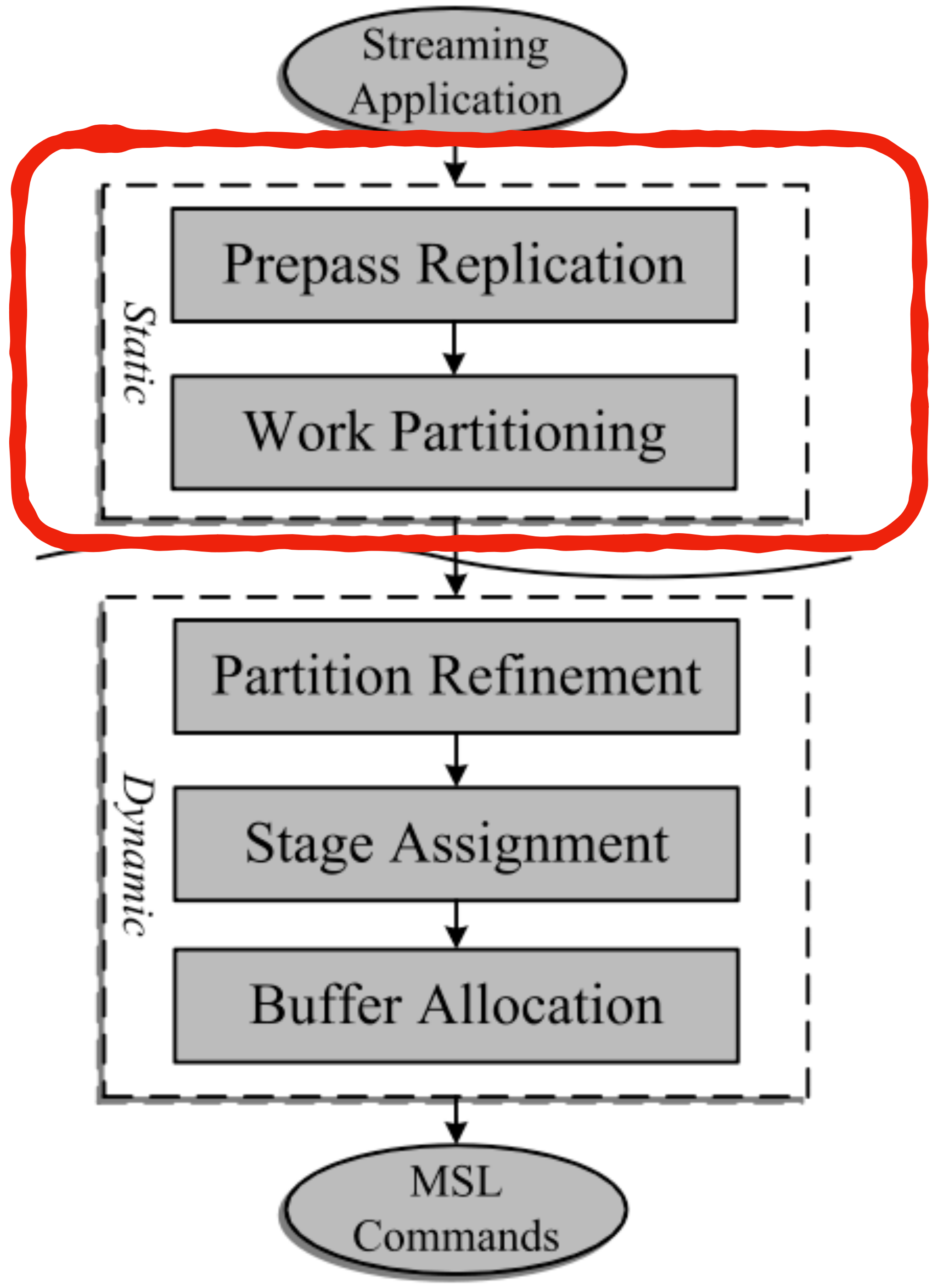
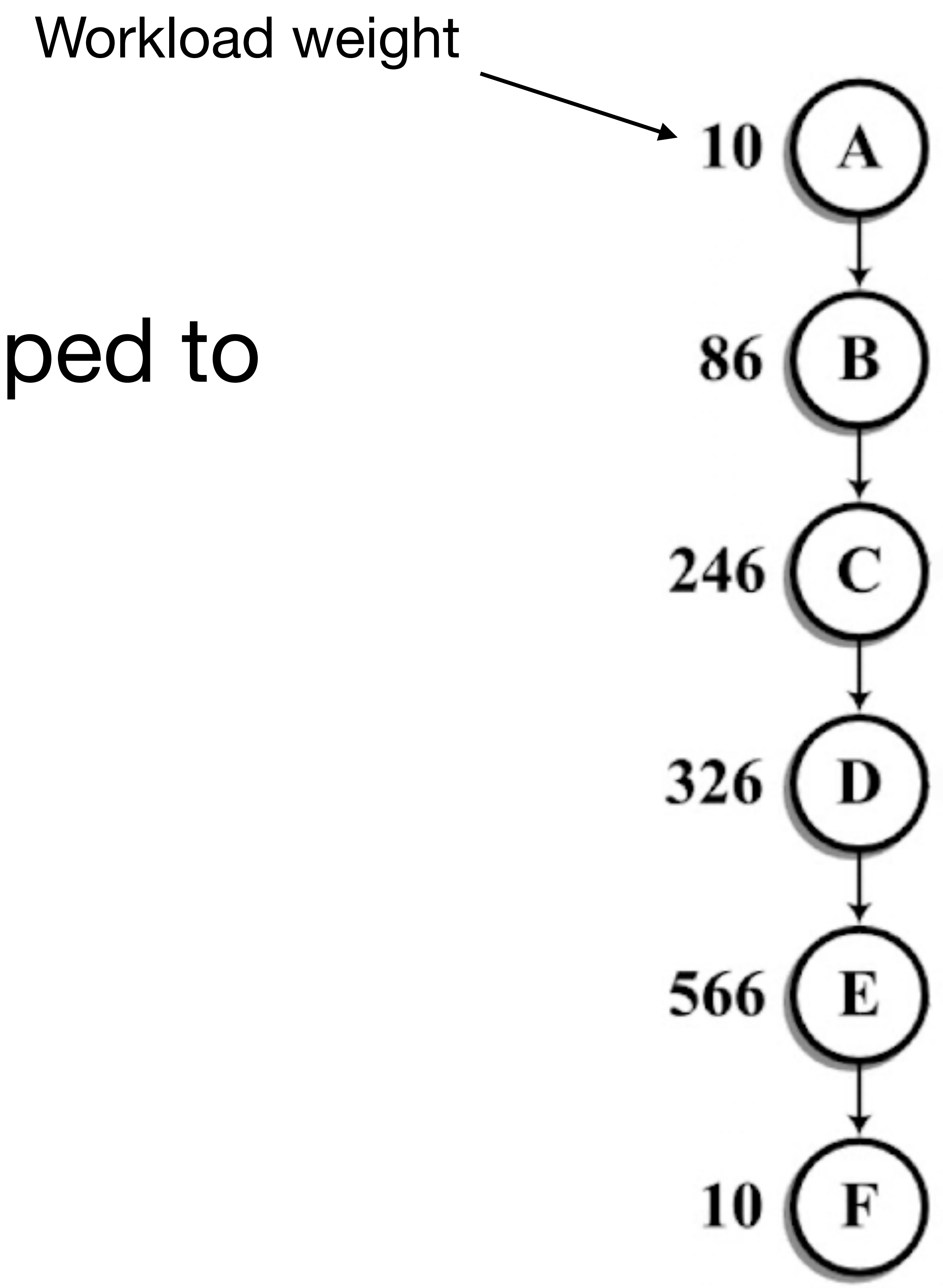


Figure 2: General flow of the Flextream framework

Prepass Replication

- Suppose we have 8 processors.
- We can partition the graph where each node is mapped to a single processor.

Processor	Compute	Work Weight
P0	A	10
P1	B	86
P2	C	246
P3	D	326
P4	E	566
P5	F	10
P6		0
P7		0

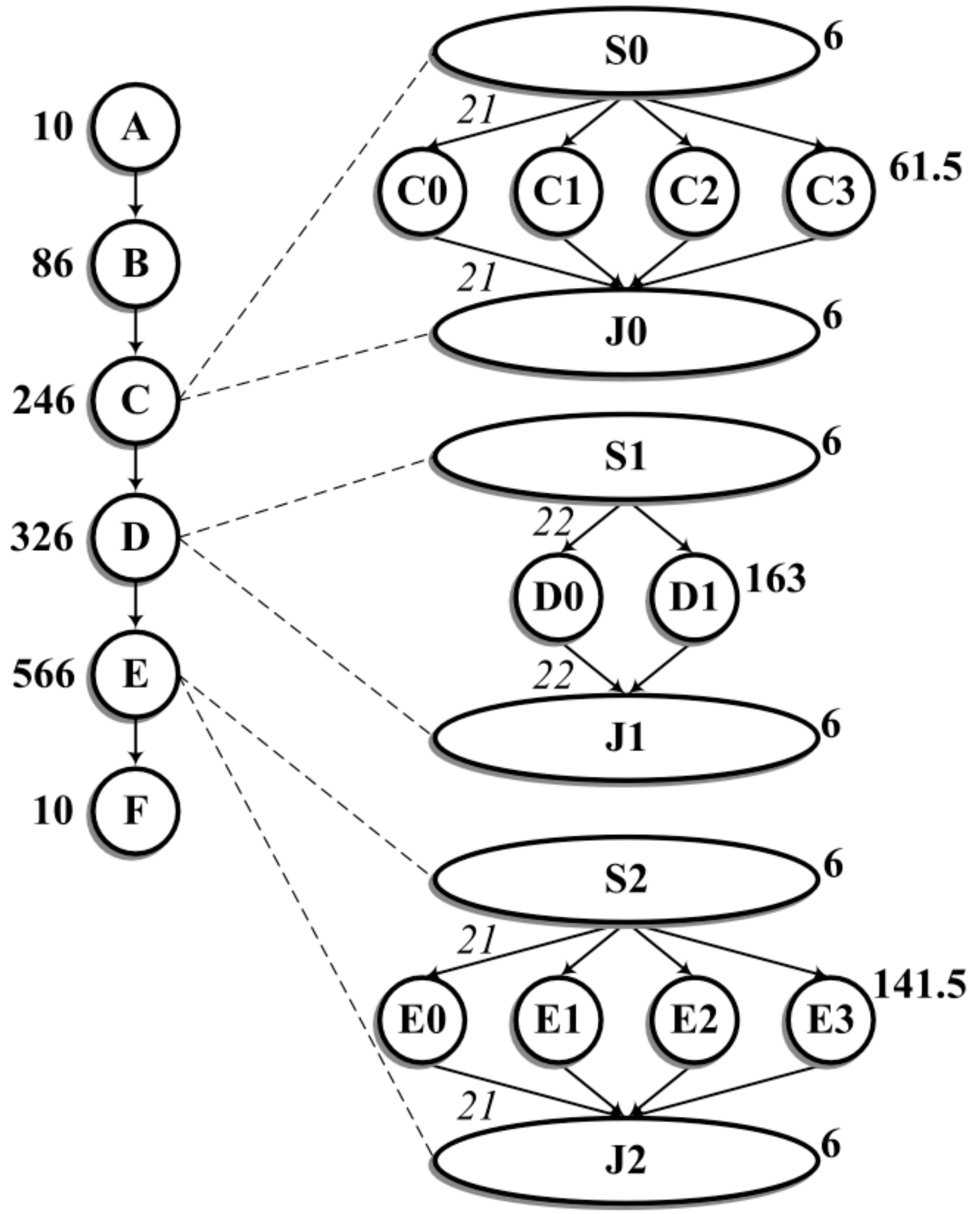


Example Stream Graph

- **Not efficient!**

Prepass Replication

- Replicate larger workloads and split it.
  - Algorithm performs an initial partitioning.
  - Then tries to balance the partitions.



Example Stream Graph After Prepass Replication

Do until:

```
maxPartition < minPartition * balanceFactor
```

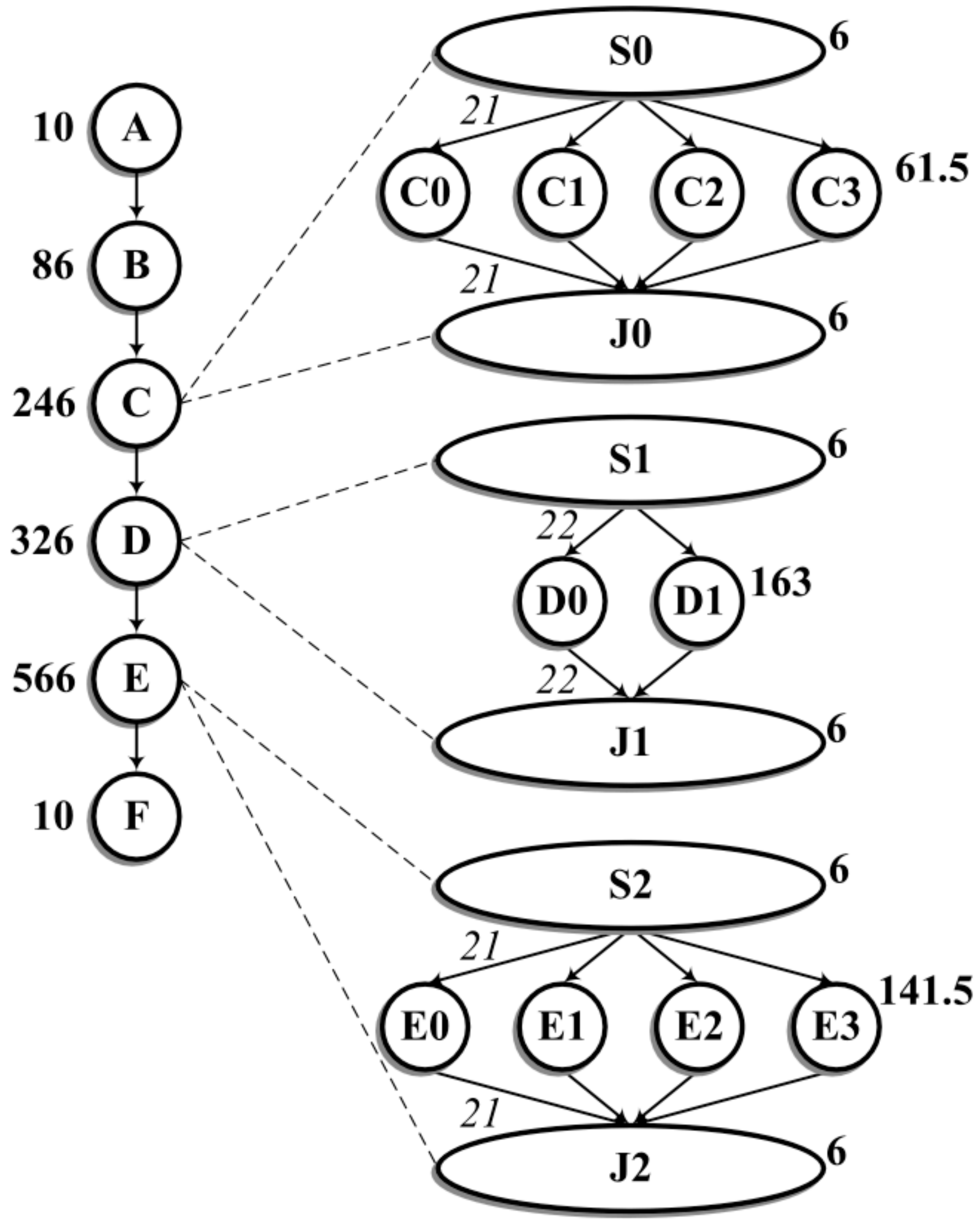
Prepass Replication

Before Prepass Replication

Processor	Compute Node	Work Weight
P0	A	10
P1	B	86
P2	C	246
P3	D	326
P4	E	566
P5	F	10
P6		0
P7		0

After Prepass Replication

Processor	Compute Node	Work Weight
P0	A, E0	151.5
P1	B, C0	147.5
P2	C1,C2,C3	184.5
P3	D0	163
P4	E1	141.5
P5	F, E2	151.5
P6	E3	141.5
P7	D1	163



Example Stream Graph After Prepass Replication

Work Partitioning

- Actual mapping of the actors to the processors.
- Solve by Integer Linear Programming (ILP) with constraints:
  - computational power of processors
  - bandwidth of interconnect
  - amount of on-chip memory
- Borrow term "initiation interval (II)" to denote inverse of throughput.

Work Partitioning

$a_{ij} = \{0, 1\}$: Indicates if actor i is running on processor j

$b_{i_1 i_2 j} = \{0, 1\}$: This variable will be 1 if connected actors (producer-consumer) i_1 and i_2 are both assigned to processor j

$$\sum_{j=0}^{P} a_{ij} = 1, \qquad \forall i$$

Actor must be mapped to only one processor.

$$\sum_{i=0}^{N} (a_{ij} \times W_i) \quad \leq \quad II \qquad \forall j$$

Workload on every processor is bounded by the II of the maximally loaded processor.

Work Partitioning

$a_{ij} = \{0,1\}$: Indicates if actor i is running on processor j

$b_{i_1 i_2 j} = \{0,1\}$: This variable will be 1 if connected actors (producer-consumer) i_1 and i_2 are both assigned to processor j

$$\sum_{(i_1\ i_2)}^{|E|} ((a_{i_2 j} - b_{i_1 i_2 j}) \times D_{i_1 i_2}) \quad \leq \quad II \qquad \forall j$$

DMA transfer workload on every processor is bounded by the II of the maximally loaded processor.

$$\sum_{(i_1,i_2)}^{|E|} [(2a_{i_1 j} + 2a_{i_2 j} - 3b_{i_1 i_2 j}) \times Buff(i_1, i_2)] \leq Mem_j, \qquad \forall j$$

The amount of buffering between two connected actors is set by the processor's memory.

Online Adaptation

Online Adaptation

- Using the optimal schedule found in static compilation, adjust schedule to the actual system.
- During runtime, if resources change, adjust mapping.

    1. **Partition Refinement:** adjust mapping to load balance
    2. **Stage Assignment:** enforce data dependence and overlap DMA
    3. **Buffer Allocation:** find where to place buffers needed between two connected actors

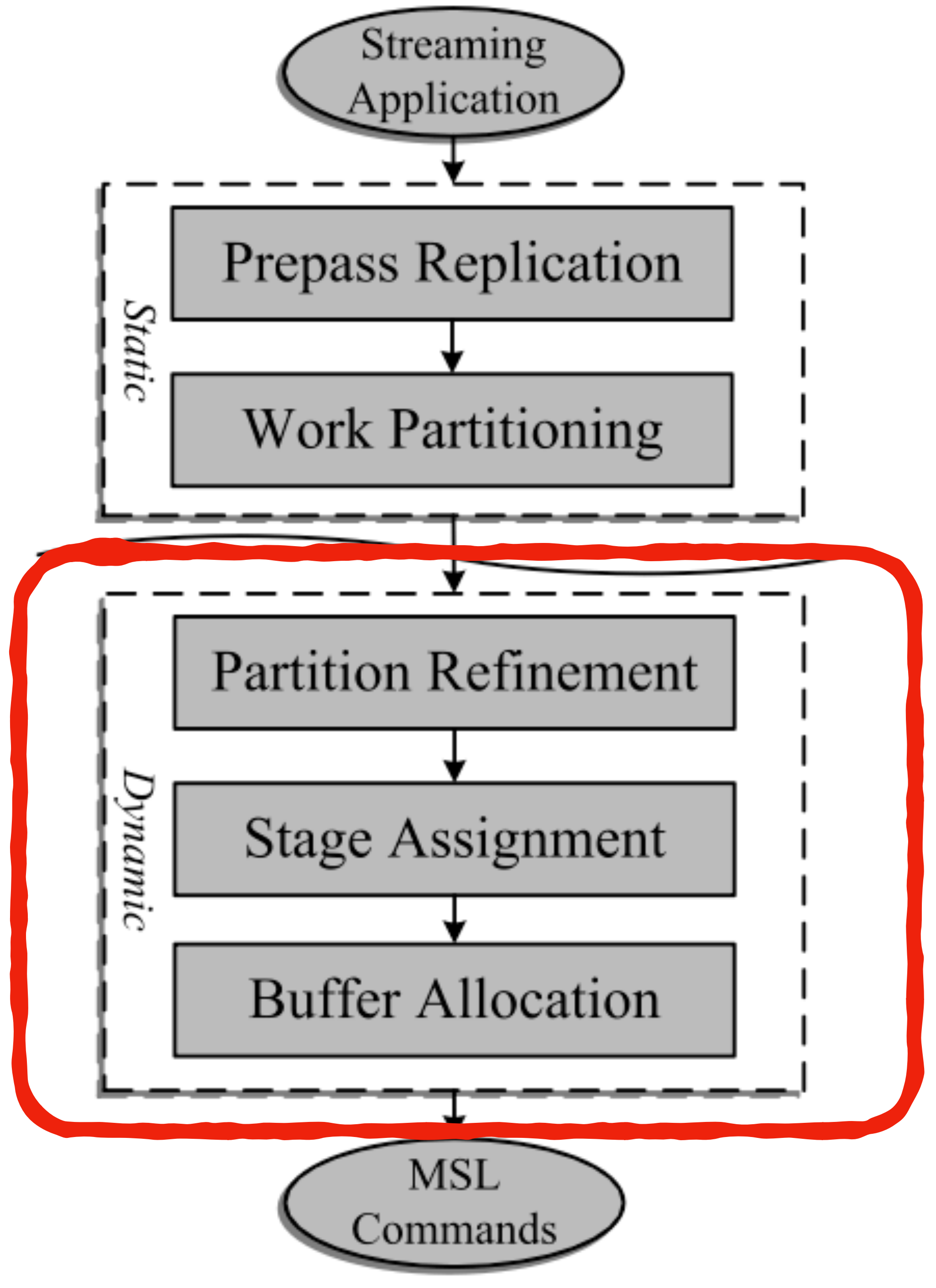


Figure 2: General flow of the Flextream framework

Partition Refinement

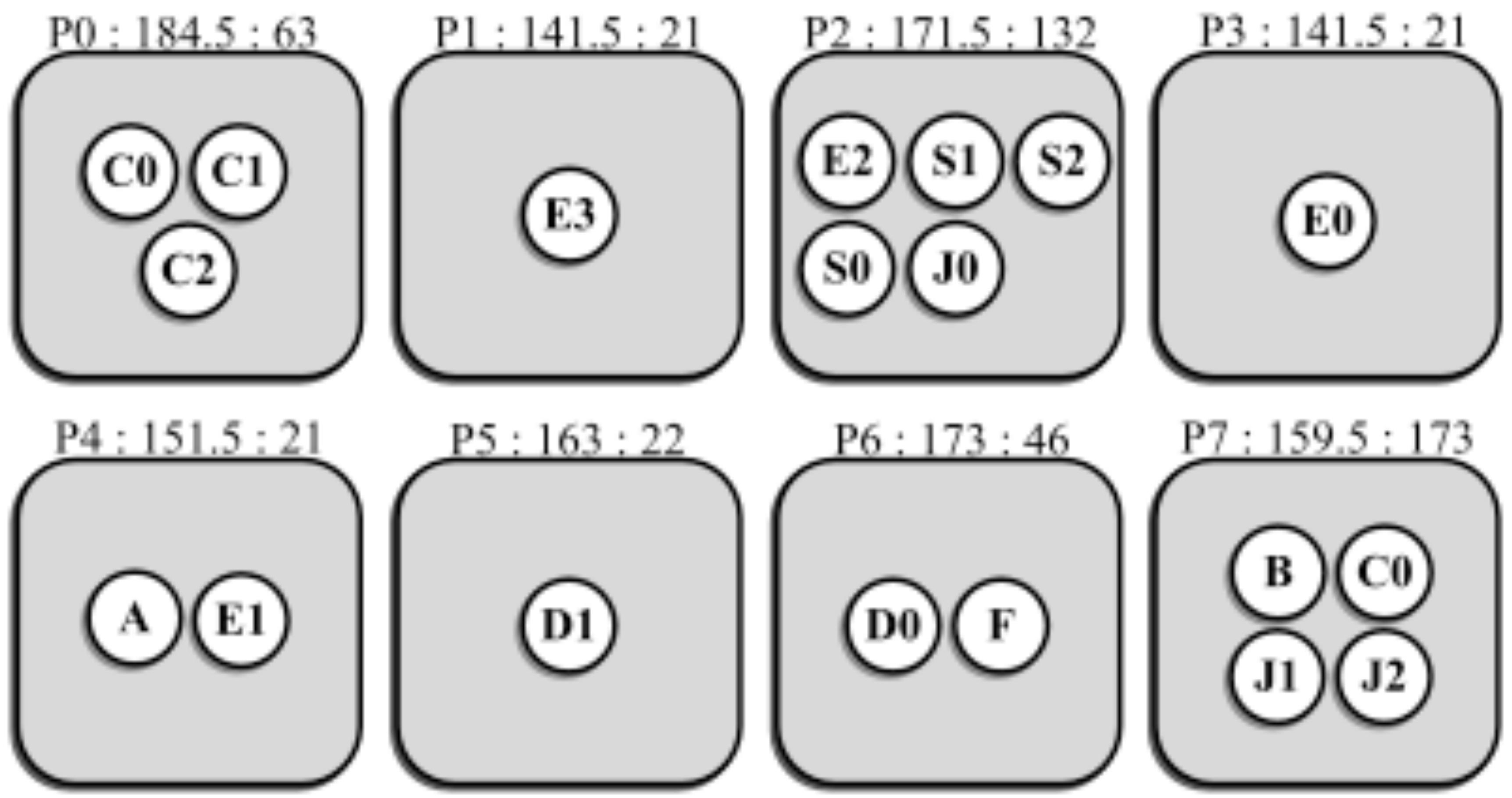
- During runtime, amount of available resources might change.
- Solving this problem is another ILP optimization problem, so a heuristic-based approach is taken.
- Refinement uses the optimal schedule found in static compilation stage to achieve a schedule close to optimal.
- Idea:
  - Try to evenly distribute workload to a new configuration.

Partition Refinement Algorithm

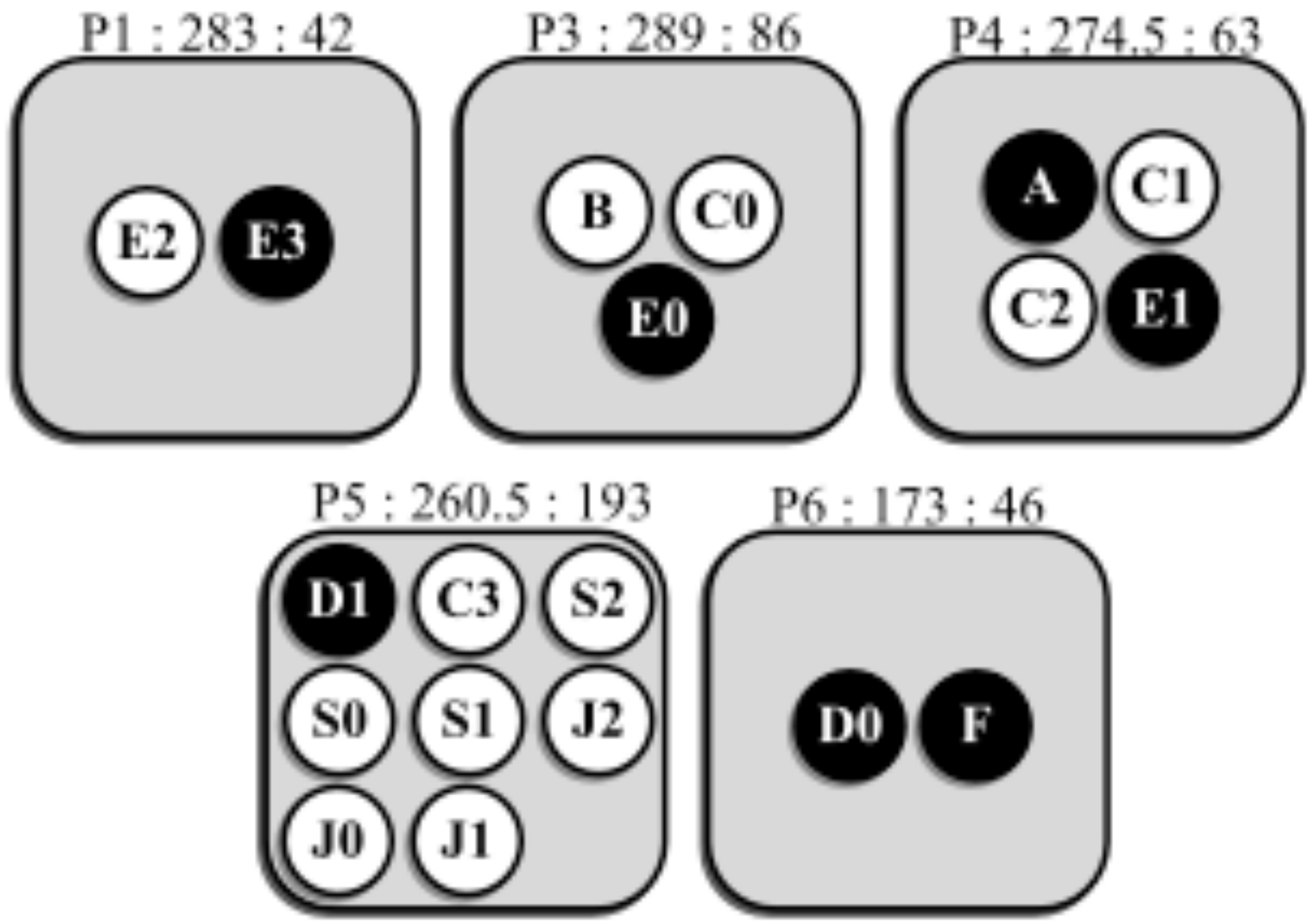
- Suppose the number of processors changed from *n* to *m* (n > m). Or, after static compilation, the actual usable processors are less than the ideal.

1. Start with *m* processors that has the least amount of scheduled actors.
2. For the remaining actors in *(n - m)* processors, sort by workload weight, put on a list.
3. Sort the *m* processors by descending workload weight.
4. Using a loop, try to **fit as many actors to one processor** until the processor reaches a specified workload threshold.

Partition Refinement Example



Static Compilation Schedule



Partition Refinement Schedule

P3 has II of 289 (3% off compared to ILP result of 283)

Stage Assignment

- Partition refinement greedily balances workload across processors.
- **But it does not consider data dependence and data transfer between two processors.**
- A producer (P) might be scheduled to a processor, and have its consumer (C) to another. This mapping may not be optimal.
- To realize throughput, actor executions corresponding to a single iteration of the entire stream graph are grouped into ***stages.***
- The main goal is to overlap DMAs between actors if they are in two different processors.

Stage Assignment

- Consider a stream graph G = (V,E).
    - Let S_i be the stage assigned to an actor i.
    - Let p_i be the processor where i is assigned.
- Rules that enforce data dependence and DMA overlap.
    1. $(i_1, i_2) \in E \implies S_{i_2} \geq S_{i_1}$ (the stage of consumer must come after producer)
    2. If $(i_1, i_2) \in E$ and $p_{i_1} \neq p_{i_2}$, then a DMA (separate stage) has to be made: $S_{i_1} < S_{DMA} < S_{i_2}$

Stage Assignment Example

Suppose (P) and (C) are in different processors.

No separate DMA stage

Iter	Producer	Consumer
k	P(k)	...
k+1	P(k+1)	DMA P(k), C(k)
k+2	P(k+2)	DMA P(k+1), C(k+1)
k+3	...	DMA P(k+2), C(k+2)

C needs to wait for DMA to finish before calculation.

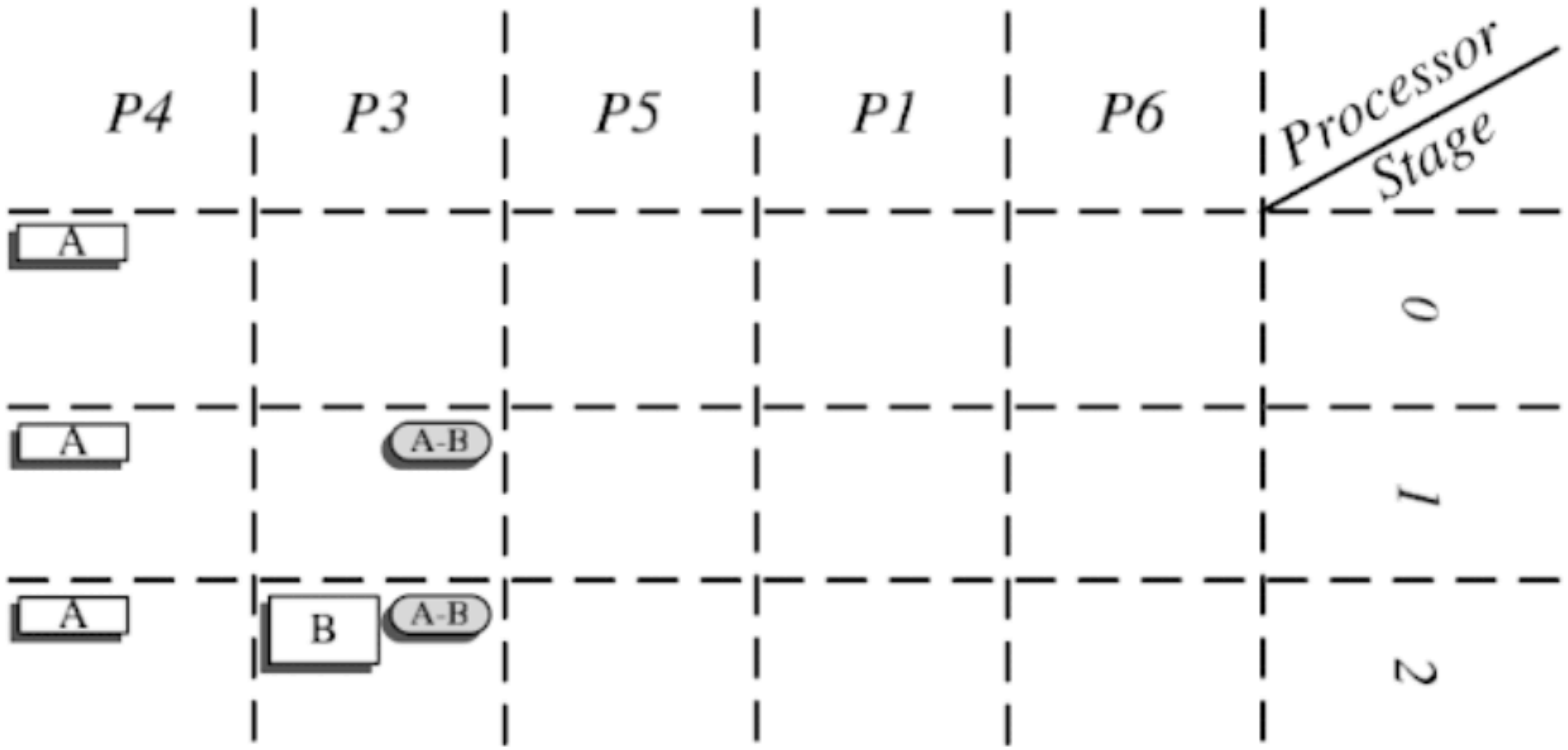
After Stage Assignment

Iter	Producer	Consumer
k	P(k)	...
k+1	P(k+1)	DMA P(k)
k+2	P(k+2)	C(k), DMA P(k+1)
k+3	...	C(k+1), DMA P(k+2)

DMA can now overlap efficiently!

Buffer Allocation

- Connected actors communicate through a set of buffers.
- Number of buffers depend on the stage number of consumer and producer, $Buffers = S_c - S_p + 1$.
- Producer A executed three times before consumer B is executed.
    - A needs 3 buffers for 3 results it produced.
- *Buffer group*: buffer between two connected actors.



Buffer Allocation

- Previous stages assume that buffers all fit in local memory of processors.
- But, depending on the amount of buffering needed, some buffers might have to be spilled to main memory.
    - This can reduce the throughput.

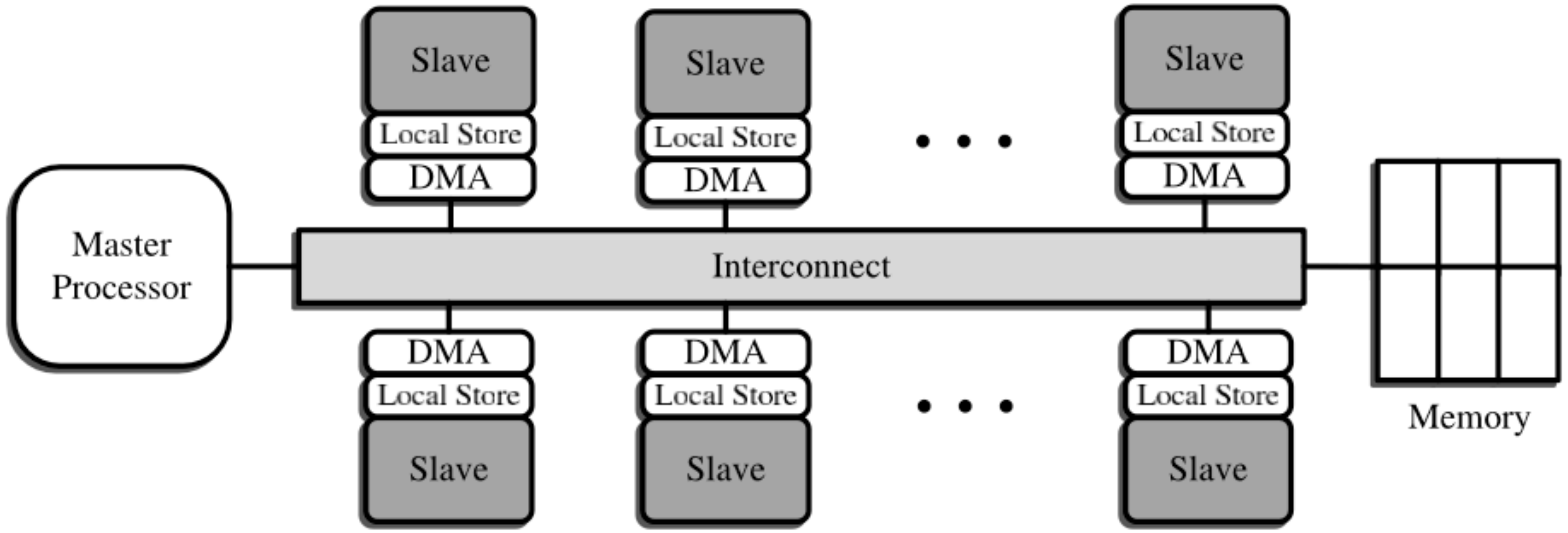
Buffer Allocation Algorithm

1. Memory usage of current schedule is calculated.
2. Make a list of victim processors that exceed the size of their local stores.
3. Make a list of victim buffer groups that does not fit in the local store of their processor.
4. For every victim buffer group, try to fit it on other non-victim processor's local store. If it does not fit, spill it to main memory.
5. After finding the new mapping, add DMAs on the schedule (update the stage assignment).

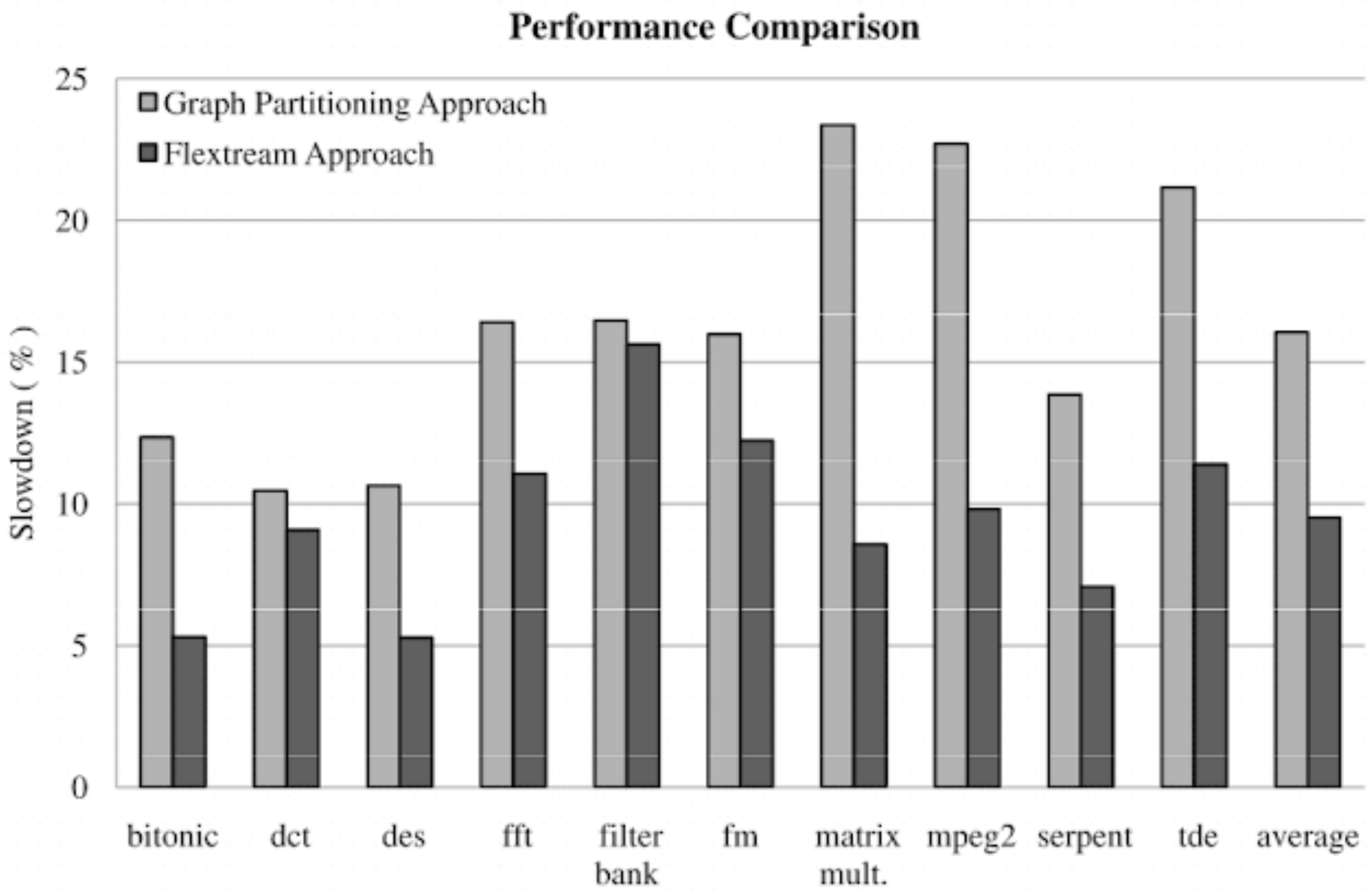
Evaluation

Evaluation

- **Comparing system**: Graph Partitioning Approach (Metis)
- **Benchmark**: StreamIt benchmark suite
- **System**: multicore system with 32 slave cores and one master core.

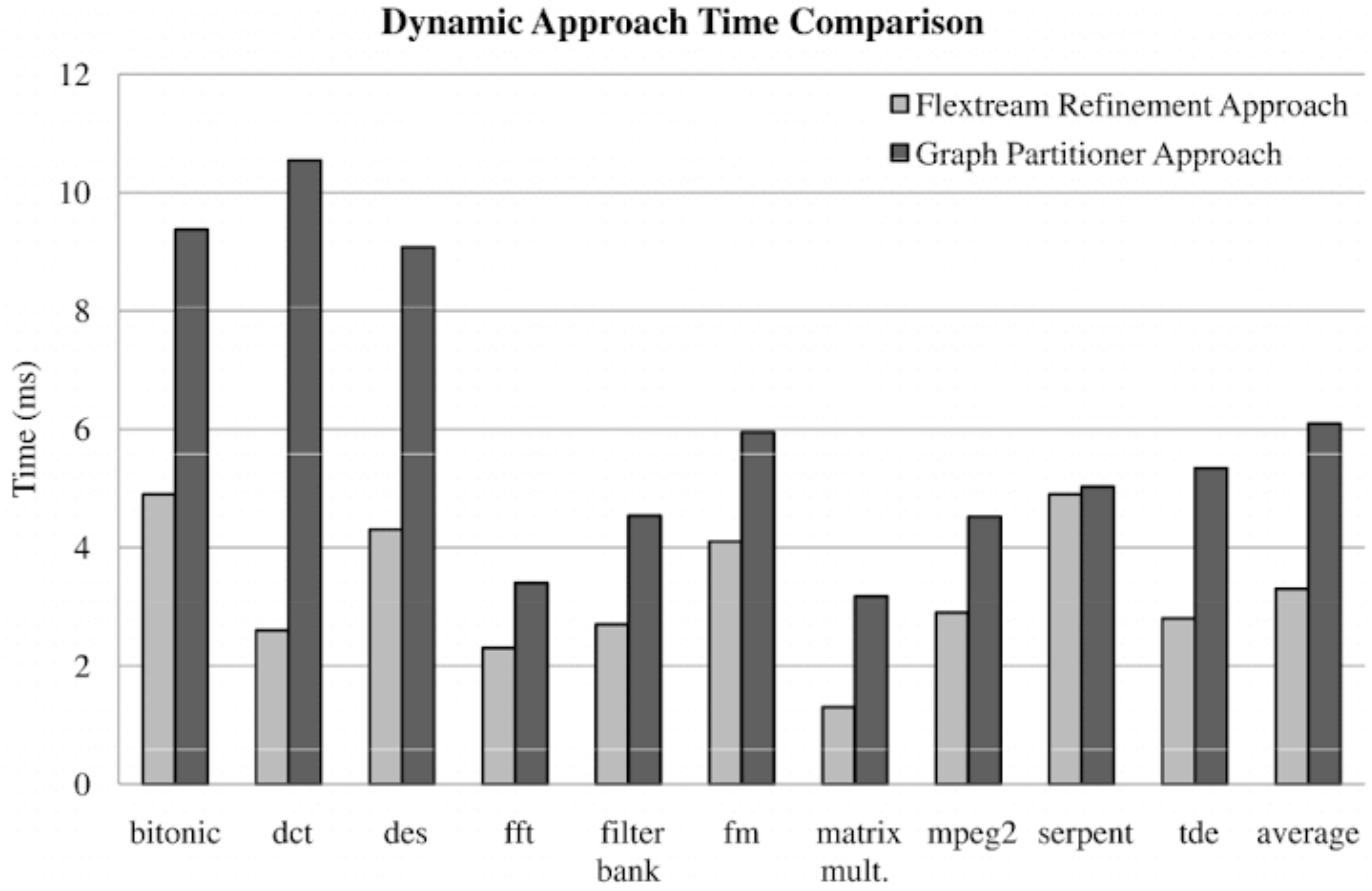


Performance Compared to ILP Solution



- Flextream's performance edge due to leveraging of optimal scheduling solution found in static compilation phase.

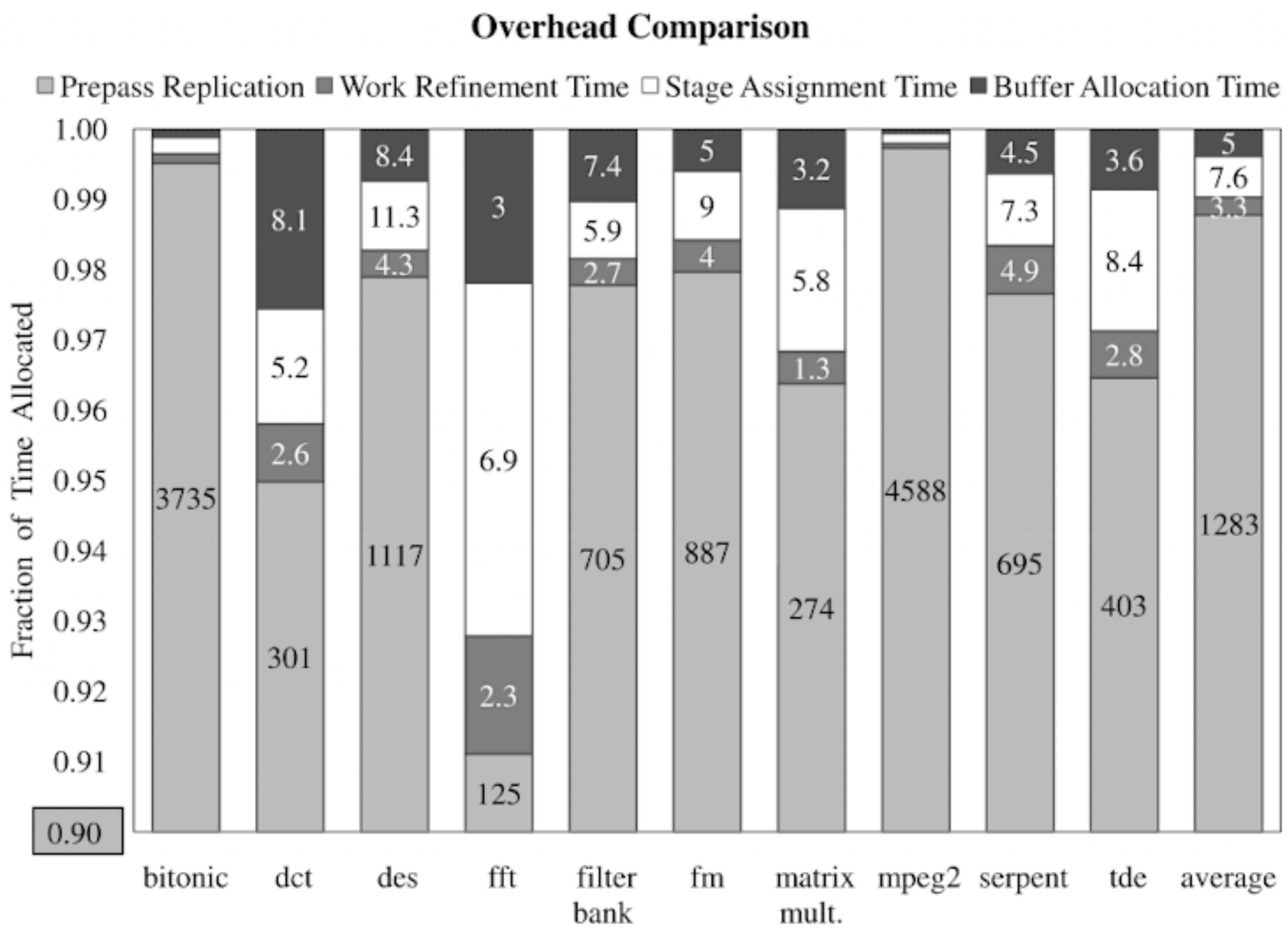
Dynamic Time Comparison



- Flextream partition refinement time is lower since it starts by using the optimal schedule found in static compilation phase.

Overhead Comparision

- Most of the overhead are in prepass replication (static compilation step).
- About 90% of the overhead is in this static compilation step.
- This shows that online adaptation is an efficient endeavor.



Throughput vs Static-only Approach

	Drain(ms)	Adaptation(ms)	1K sec-Flextream	1K sec-Static
bitonic sort	6.14	89.42	350M	356M
dct	0.79	42.80	380 M	452 M
des	32.39	113.80	148 M	150 M
fft	2.37	142.95	222 M	230 M
filter bank	0.44	142.95	448 M	490 M
fm	2.16	65.71	133 M	143 M
matrix mult.	3.07	37.19	62 M	71 M
mpeg2	4619	43	156 K	177 K
serpent	81.11	79.09	52 M	54 M
tde	780	66.08	1.2 M	1.3 M

- **Drain time:** time to remove the current schedule.
- **Adaptation time:** drain time + time to communicate the new schedule

Throughput vs Static-only Approach

	Drain(ms)	Adaptation(ms)	1K sec-Flextream	1K sec-Static
bitonic sort	6.14	89.42	350M	356M
dct	0.79	42.80	380 M	452 M
des	32.39	113.80	148 M	150 M
fft	2.37	142.95	222 M	230 M
filter bank	0.44	142.95	448 M	490 M
fm	2.16	65.71	133 M	143 M
matrix mult.	3.07	37.19	62 M	71 M
mpeg2	4619	43	156 K	177 K
serpent	81.11	79.09	52 M	54 M
tde	780	66.08	1.2 M	1.3 M

- Last to columns show how many iterations of each stream graph can be executed; ie, throughput.
- Flextream achieves about the same throughput as the theoretical optimal solution.

Conclusion

- Flextream is a flexible compilation framework that can dynamically adapt applications to the changing characteristics of the underlying architecture.
- Static compilation phase finds the optimal schedule assuming all resources are available.
- Dynamic adaptation starts with this optimal solution to fit the schedule based on the actual available resources.
- Results show that Flextream performs relatively close to the theoretical optimal solution.