Chapter 6-1: Multiprocessor Software

Soo-Ik Chae

Multiprocessor software

- Performance analysis of multiprocessor software
- Middleware and software services
- Design verification of multiprocessor software

- Multiprocessor: true concurrency
  - Single processor: virtual concurrency
- Hard to analyze and debug

Topics

- Performance analysis of multiprocessor software.
    - Models.
    - Analysis.
    - Simulation.

What is different about embedded multiprocessor software?

- How does it differ from general-purpose multiprocessor software?
- How does it differ from a uniprocessor?

Heterogeneity

- Hardware platforms are heterogeneous
- Heterogeneity presents several types of problems
  - Getting SW form several types of processors to work together can present challenges.
    - endianness
  - Development environments for heterogeneous multiprocessors are often loosely coupled.
    - Programmers may have a hard time learning all the tools for all the component processors
    - It may be hard to debug problems that span multiple CPU types.
  - Different processors may offer different types of resources and interfaces to those resources.
    - Not only does this complicate programming but it also makes it harder to decide certain things at runtime.

Delay variations

- Delay variations are harder to predict in multiprocessors:
  - Subtle timing bugs are more likely to be exposed.
  - Makes it harder to efficiently use system resources.
  - Long memory access times complicate algorithm design and programming.
- Scheduling a multiprocessor is hard---information about the state of the processors costs time, energy.
- Optimal scheduling algorithm do not exist for the most realistic multiprocessor configurations.
  - Heuristics must be used.
  - Due to communication delay, state information of other processors takes too long to get. So scheduling decision must be made with full information about other processor states.

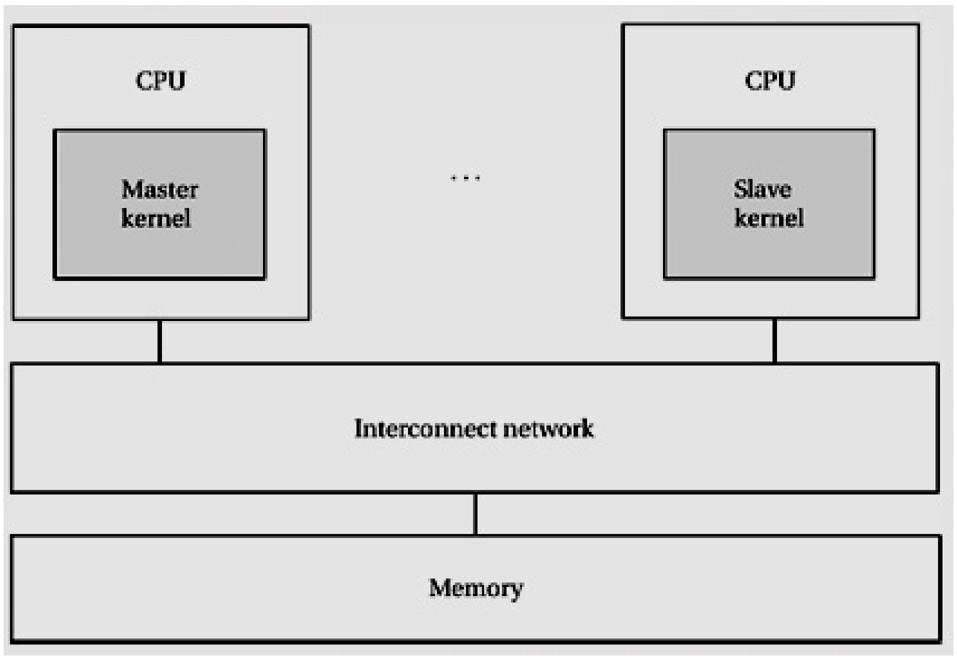
Resource allocation

- Resources must be allocated dynamically to ensure that they are used efficiently.
- Just knowing which resource are available in a multiprocessor is hard enough.
- Determining on-the-fly which resources are available in a multiprocessor is hard too.
- Figuring out how to use those resources to satisfy requests is even harder.
- Middleware takes up the task of managing system resources across the multiprocessor.

Role of the multiprocessor operating system

- Simple multiprocessor OS has one master, one or more slaves.
  - Simple to implement.
  - Suitable for symmetric multiprocessor systems
  - Heterogeneous processors limit resource allocation options.
- Each processor has its own kernel
  - Responsible for managing purely local resources such as the devices that are visible to other processors.
  - The PE kernel selects the processes to run next and switches contexts as necessary.
    - But the PE kernel may not decide entirely on its own which process runs next.
  - It may receive instruction from a kernel running on another processor

Kernels in the multiprocessor

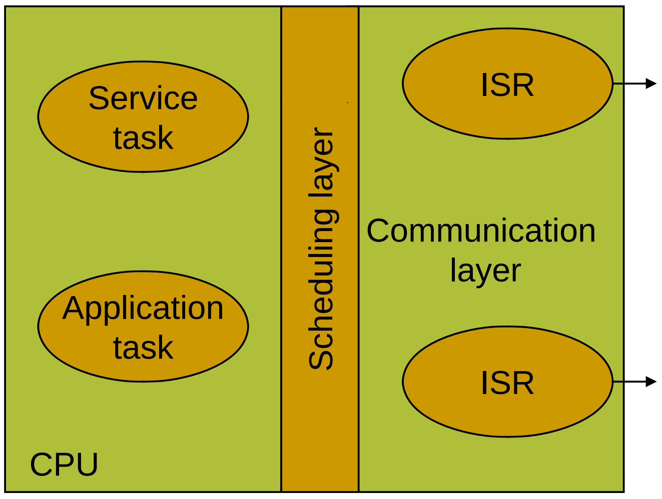


Limited scheduling information

- The master kernel gathers information from the slave PEs
- Based on the current state of the slaves and the processes that want to run on slaves, the master kernel then issues commands to the slaves about their schedules.
- One challenge in designing distributed schedulers is that
  - communication is not free and
  - any processor that makes scheduling decisions about other PEs usually will have incomplete information about the state of that PE.
- When a kernel schedules its own processor, it can easily check on the state of that processor.
- When a kernel must perform a remote read to check the state of another processor, the amount of information the kernel requests needs to be carefully budgeted.

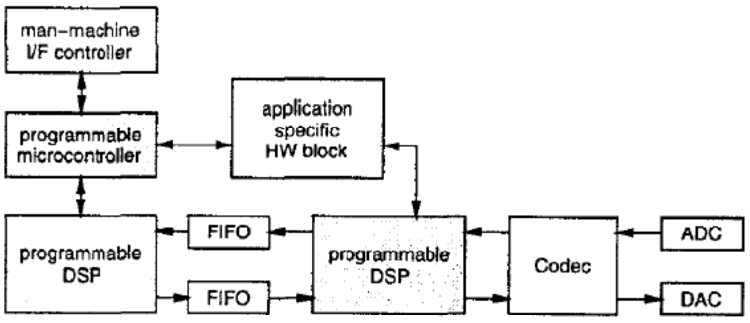
Kernel architecture (Vercauteren)

- A kernel architecture for custom heterogeneous processors includes scheduling and communication layers.
- Basic communication operations implemented by interrupt service routines.
- Kernel channel used only for kernel-to-kernel communication.
  - Optimized for performance
- Data channel is used by applications
  - More general purpose



Multiprocessor systems

- No tool support for heterogeneous embedded system architecture
- Should provide real-time kernel support for managing the current software tasks that are distributed over several processors



man-machine I/F controller
programmable microcontroller
application specific HW block
programmable DSP
FIFO
FIFO
programmable DSP
Codec
ADC
DAC

Fig. 1. Heterogeneous embedded system architectures.

Target architecture model

- Communication channels
  - Semantics: Hoare's CSP rendezvous
  - Explicit send and receive , or
  - Shared memory
- Hardware components
  - Parameterized communication components
  - Hardware processors
  - Memory components
- Software components
  - Processor + Icache+ Dcache + I/O units (wrappers)

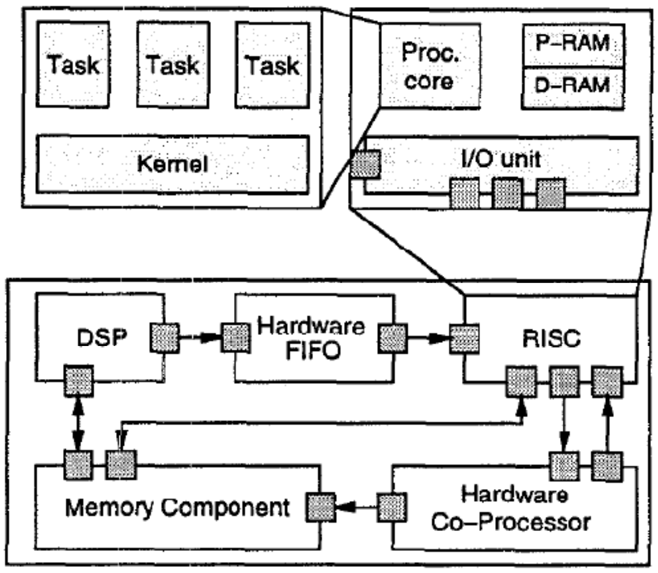


Fig. 2. Target Architecture Model.

Basic kernel architecture

- Kernel is responsible for
  - Scheduling application tasks
  - Handling communication between application tasks
  - Synchronizing the application tasks with each other and with external events
  - Preemptive, priority-driven scheduling

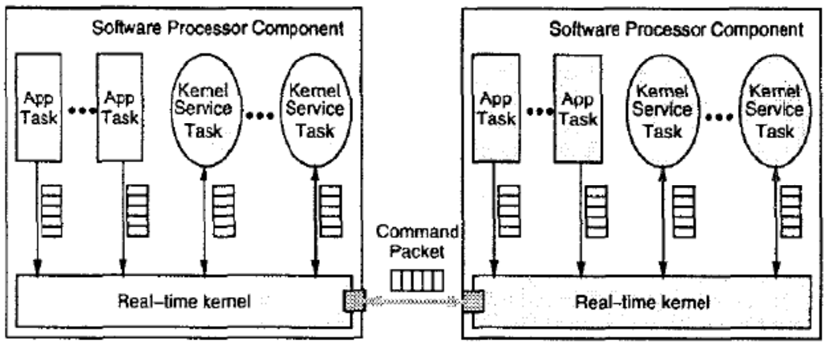


Fig. 3. Basic Kernel Architecture

Basic kernel architecture

- Kernel also provides a subroutine interface to each predefined kernel service task
  - Resource protection
  - Memory (de)allocation
  - Communication and synchronization between application tasks.

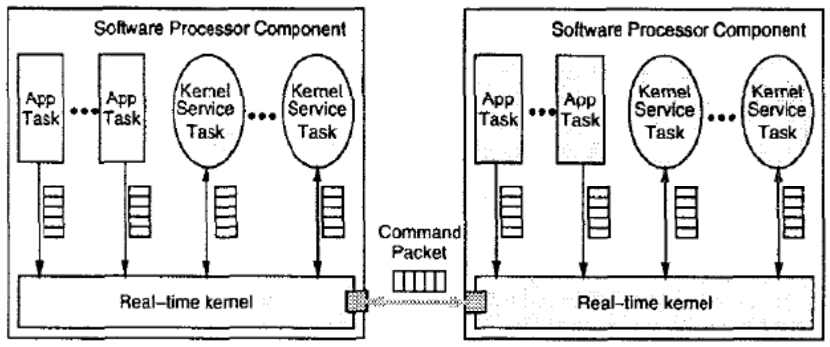


Fig. 3. Basic Kernel Architecture

Basic kernel architecture

- Kernel channel
- Data channel

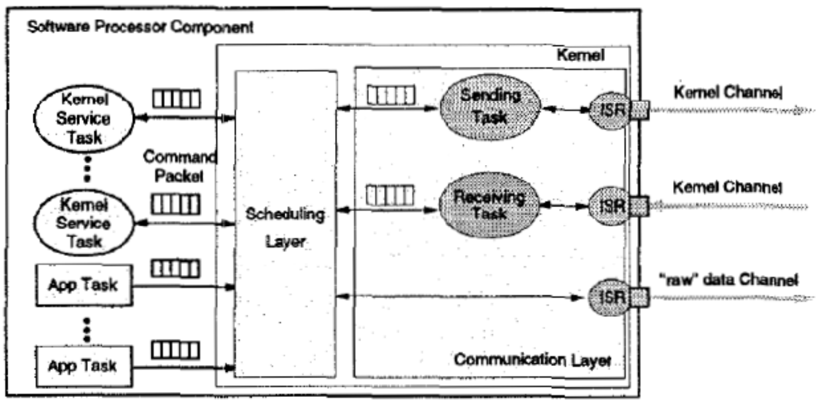
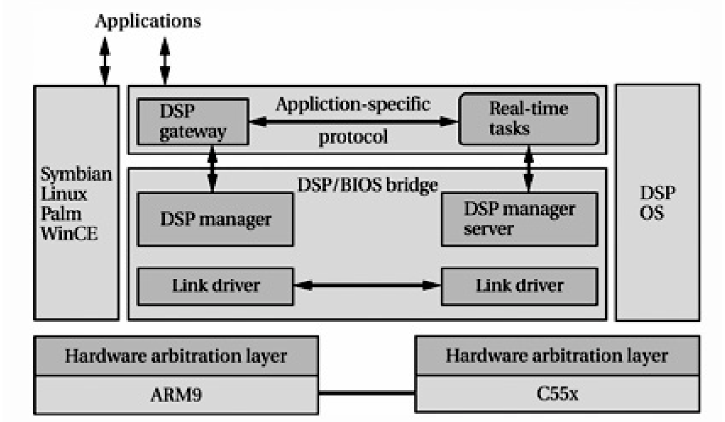


Fig. 4. Kernel architecture with Communication Layer expanded

OMAP lower layers including HW and OS

- The main unifying structure in OMAP is the DSPBridge, which allows the DSP and RISC processors to communicate.
- The bridge includes a set of hardware primitives that are abstracted by a layer of software.
- The bridge is organized as a master/slave system in which the ARM is the master and the C55x is the slave.



OMAP lower layers including HW and OS

- This master/slave system fits the nature of most multimedia applications, where
  - DSP is used to efficiently implement certain key functions
  - while RISC processor runs the higher levels of the application.
- The DSPBridge API implements several functions:
  - initiates and controls DSP tasks,
  - exchanges messages with the DSP,
  - streams data to and from the DSP, and
  - checks the status of the DSP.
- OMAP hardware provides mailbox primitives - separate addressable memories that can be accessed by both.
  - In the OMAP 5912, two mailboxes can be written only by the C55x but read by both,
  - other two can be written only by the ARM and read by both.

Mailbox primitives

- **send** (*A, message*)
  - send a *message* to mailbox *A*
- **receive** (*A, message*)
  - receive a *message* from mailbox *A.*

OMAP C5510 performance/power for AAC decoding (from TI)

Rate	Mcycles/ sec	mA @ 1.5V	mA @ 1.2V
64K	22.1	8.0	6.4
48K	16.2	5.8	4.7
32K	11.4	4.1	3.3

Multiprocessor scheduling (Stone)

- It is rather allocation problem
- Schedule tasks on two CPUs.
  - Actually allocates tasks to the CPUs to satisfy scheduling constraint.
- General scheduling problem is NP-complete
- By using information of the multiprocessor structure, or by simplification, this problem can be solved in polynomial time.
  - Exact solution for two processors.
  - Heuristics for more processors.
- Solve using network flow algorithms.

Multiprocessor modeling (Stone)

- Execution time table provides execution time of processes on the two CPUs.
- Intermodule connection graph describes the time cost of communication between two processes when they run on different CPUs.
  - Communication time within a CPU is zero.
- Modify intermodule communication graph:
  - Add two additional nodes:
    - source node for CPU 1 and sink node for CPU 2.
  - Add edges from each non-sink node to source and sink.
    - Edge weight to source is cost of executing on CPU 2 (sink).
    - Edge weight to sink is cost of executing on CPU 1 (source).
- Minimize total time by finding a minimum-cost cutset of the modified intermodule connection graph.

Stone multiprocessor example 1

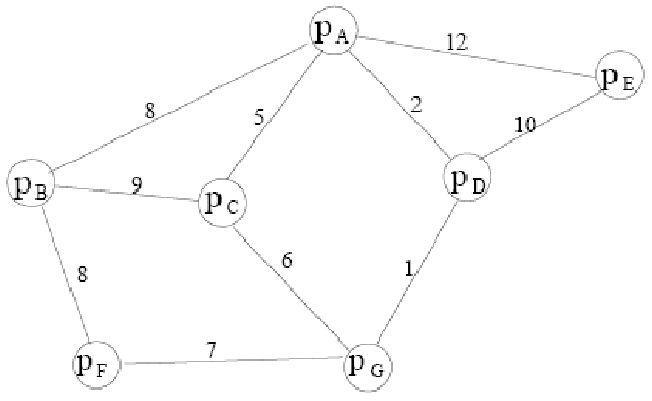
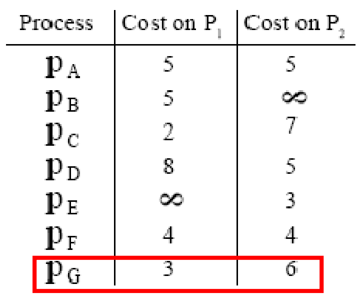


Figure 17: Process communication model

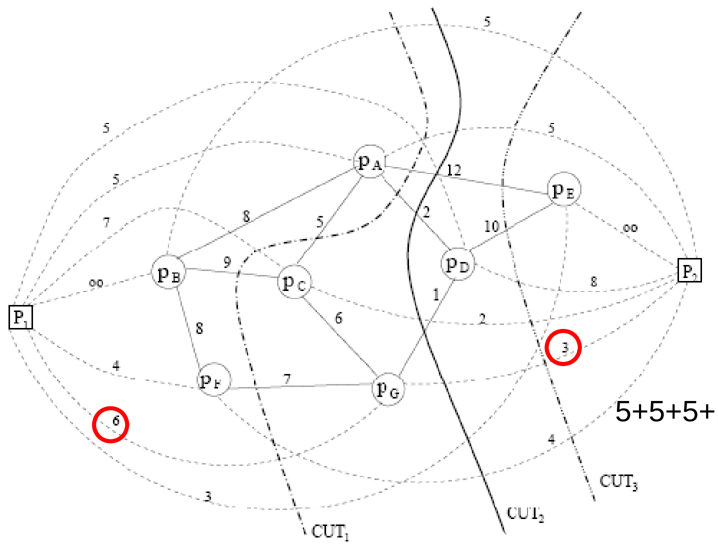


Process	Cost on P_1	Cost on P_2
p_A	5	5
p_B	5	∞
p_C	2	7
p_D	8	5
p_E	∞	3
p_F	4	4
p_G	3	6

Figure 18: Process execution cost

- Execution time table
- Intermodule connection graph

Stone multiprocessor example 1

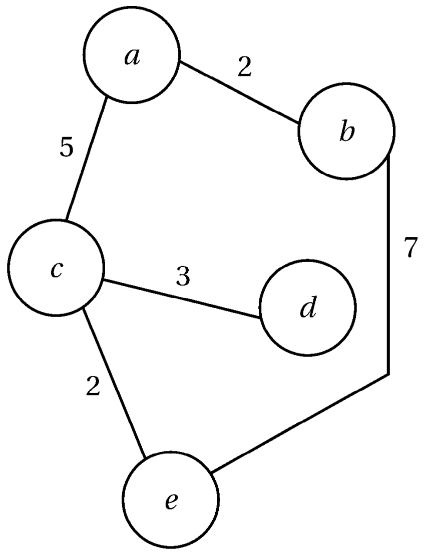


5+5+5+12+2+2+1+3+3+4=42

In Stone's flow network, any cut defines a valid assignment of processes to processors. Moreover, the value of the cut is equal to the cost of the assignment it defines. For example in Fig. 19, three possible cuts are presented. The values of the cuts are: $CUT_1 = 74$, $\boxed{CUT_2 = 42}$, and $CUT_3 = 52$. The minimum cut is CUT_2. It assigns processes p_A, p_B, p_C, p_F, and p_G to processor P_1, and processes p_D and p_E to processor P_2. Therefore the two-processor module allocation problem reduces to determining the minimum cut in a flow network.

? (B,C,F,G)(A,D,E): 5+5+5+8+5+2+1+3+3+4=41

Stone multiprocessor example 2



Intermodule connection graph

Process	CPU 1 runtime	CPU 2 runtime
a	8	7
b	5	–
c	12	15
d	3	6
e	4	4

Execution time table

Why static tasks?

- Many embedded systems statically allocate processes to processing elements.
- We can efficiently find bounds on the execution time of the processes in those multiprocessor systems.
- Static task allocation determines allocation to CPU at design time.
- Static task allocation reduces OS overhead, allows more analysis.
- Dynamic task allocation can choose the CPU for a task at run time.
- Dynamic task allocation helps manage dynamic loads.

Synchronous Data Flow (SDF)

- In SDF a program is represented as a directed graph in which vertices, which are called actors, represent computations, and the edges specify FIFO channels for communication between actors.
- The term "synchronous" refers to the requirement that the number of data values produced (consumed) by each actor onto (from) each of its output (input) edges is a fixed value for each firing of that actor and is known at compile time.
- It should not be confused with the use of "synchronous" in the synchronous languages.

Synchronous languages

- A change in the state of one module is simultaneous with receipt of inputs.
- Outputs from a module are simultaneous with changes in state.
- Communication between modules is synchronous and instantaneous.
- Output behavior of the modules is entirely determined by the interleaving of input signals.

Synchronous languages

- Imperative: Esterel, SyncCharts
  - Provide constructs to shape control-dominated programs as hierarchical synchronous automata.
- Declarative: Lustre, Signal
  - Shape applications based on intensive data computation and data-flow organization, with the control flow operating under the form of (internally generated) activation clocks.

Synchronous hypothesis

- Is really a collection of assumptions of a common nature, sometimes adapted to the framework considered.
- Instants and reactions: In each instant, input signals possible occur (for instance by being sampled), internal computation take place, and control and data are propagated until output values are computed and a new global system state is reached.
  - This execution cycle is called reaction of the system to the input signals. Reactions converge and computations are entirely performed before the current execution instant ends and a new one begins.
  - This empowers the obvious conceptual abstraction that computations are infinitely fast (instantaneous, zero-time), and take place only at discrete points in (physical) time. With no duration.

Synchronous hypothesis

- Signals: broadcast signals are used to propagate information.
  - At each execution instant, a signal can either be present or absent. A signal must be consistent for all read operations during any given instant.
- Causality: an important part of the theoretical body behind the Synchronous Hypothesis.
  - The presence status and value of a signal should be defined before they are read (and tested).
  - “before” refers to here to causal dependency in the computation of the instant, and not to physical or even logical time between successive instant.
- The Synchronous Hypothesis ensures that all possible schedules of operations amounts to the same result (convergence).

Homogeneous SDG = DFG

- Homogeneous SDF: the numbers of data values produced or consumed are identically unity.
- Data flow graph (DFG) : homogeneous SDF
- By scheduling, we collectively refer to the tasks of
  - Assigning actors in DFG to processors
  - Ordering execution of these actors on each processor
  - And determining when each actor fires

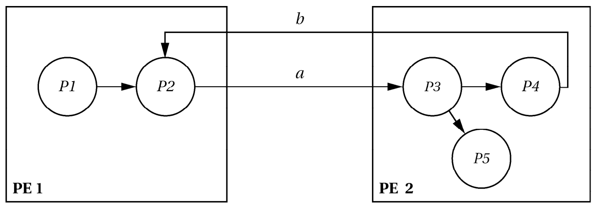
  such that data precedence constraints are met.
- In the fully static scheduling strategy, all three scheduling tasks are performed at compile time.
  - Assumes that exact execution times of actors are known

Self-timed scheduling strategy ★

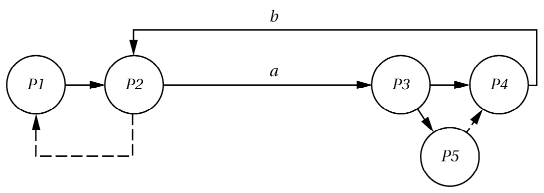
- Assumes that good estimates for the execution times of the actors can be obtained.
- Each processor executes the actors assigned to it in the order specified at compiled time.
- Before firing an actor, a processor wait for the data needed by that actor to become available.
- Thus, in self-timed synchronization processors are required to perform run-time synchronization when they communicate data.
- As a result, the self-timed strategy incurs greater run-time cost than the fully static case because of the synchronization overhead.

SDF scheduling (Bhattacharyya)

- Interprocessor communication modeling (IPC) graph has the same nodes as SDF, all SDF edges, plus additional edges.
- Added edges model sequential schedule.
  - Dashed lines in the figure
- Edges that cross processor boundaries are called IPC edges.
  - Must use an interprocessor communication mechanism



SDF graph and processor allocation

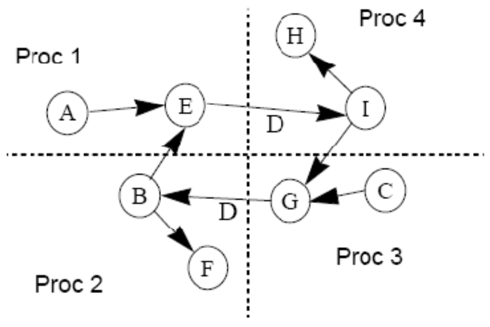


Interprocessor communication modeling graph

Strongly connected component (SCC)

- A DFG (V,E) is strongly connected if for each pair of distinct vertices x,y there is a path directed from x to y and there is a path directed from y to x.
- A strongly connected component (SCC) of (V,E) is a strongly connected subset V' $\subseteq$ V such that V properly contains V'.
- If V is an SCC, its associated subgraph is also called as SCC.
- An SCC V' of a DFG (V,E) is a source SCC
  if $\forall e \in E$, $(sink(e) \in V') \Rightarrow (src(e) \in V')$
  - Source가 V'에 있는 edge중 밖으로 나가는 것이 있다
- An SCC V' of a DFG (V,E) is a sink SCC
  if $\forall e \in E$, $(src(e) \in V') \Rightarrow (sink(e) \in V')$
  - Sink가 V'에 있는 edge중 밖에서 들어오는 것이 있다.

Self-time schedule

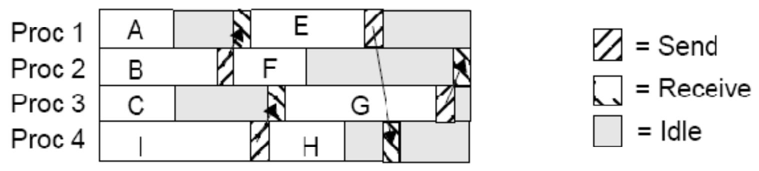


(a) DFG "G"

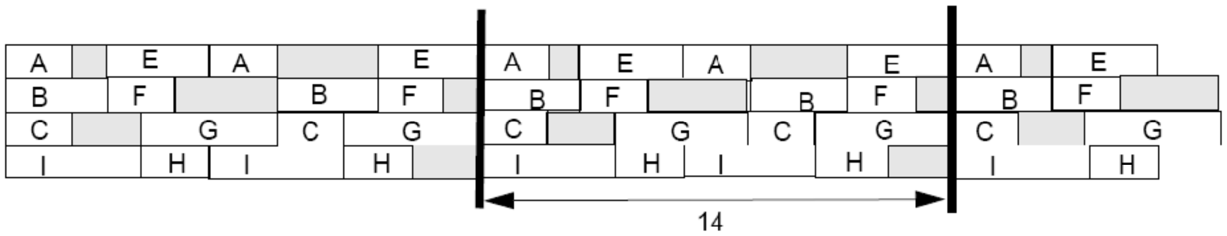
Execution Time Estimates

A, C, H, F	: 2
B, E	: 3
G, I	: 4

Self-time schedule

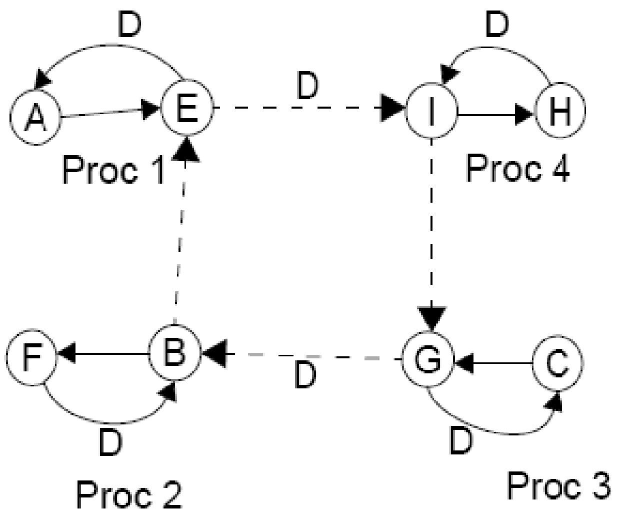


(b) Schedule on four processors



(c) Self-timed execution

Self-time schedule



(d) The IPC graph

IPC graph

The IPC graph has the same vertex set V as G, corresponding to the set of actors in G. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Fig. 1, processor 1 executes A and then E repeatedly. We model this in G_{ipc} by drawing a cycle around the vertices corresponding to A and E, and placing a delay on the edge from E to A. The delay-free edge from A to E represents the fact that the kth execution of A precedes the kth execution of E, and the edge from E to A with a delay represents the fact that the kth execution of A can occur only after the $(k-1)$ th execution of E has completed. Thus if actors $v_1, v_2, \ldots, v_n$ are assigned to the same processor in that order, then G_{ipc} would have a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $delay((v_n, v_1)) = 1$. If there are P processors in the schedule, then we have P such cycles corresponding to each processor.

IPC graph

The IPC graph has the same vertex set V as G, corresponding to the set of actors in G. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Fig. 1, processor 1 executes A and then E repeatedly. We model this in G_{ipc} by drawing a cycle around the vertices corresponding to A and E, and placing a delay on the edge from E to A. The delay-free edge from A to E represents the fact that the kth execution of A precedes the kth execution of E, and the edge from E to A with a delay represents the fact that the kth execution of A can occur only after the $(k-1)$ th execution of E has completed. Thus if actors $v_1, v_2, \ldots, v_n$ are assigned to the same processor in that order, then G_{ipc} would have a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $delay((v_n, v_1)) = 1$. If there are P processors in the schedule, then we have P such cycles corresponding to each processor.

IPC graph

The IPC graph has the same vertex set V as G, corresponding to the set of actors in G. The self-timed schedule specifies the actors assigned to each processor, and the order in which they execute. For example in Fig. 1, processor 1 executes A and then E repeatedly. We model this in G_{ipc} by drawing a cycle around the vertices corresponding to A and E, and placing a delay on the edge from E to A. The delay-free edge from A to E represents the fact that the kth execution of A precedes the kth execution of E, and the edge from E to A with a delay represents the fact that the kth execution of A can occur only after the $(k-1)$th execution of E has completed. Thus if actors $v_1, v_2, \ldots, v_n$ are assigned to the same processor in that order, then G_{ipc} would have a cycle $((v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1))$, with $delay((v_n, v_1)) = 1$. If there are P processors in the schedule, then we have P such cycles corresponding to each processor.

- The IPC graph has the same semantics as a DFG, and its execution models the execution of the corresponding self-time schedule.

Cycle Mean

Lemma 3: The asymptotic iteration period for a *strongly connected* IPC graph G when actors execute as soon as data is available at all inputs is given by [28]:

$$T = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\}. \tag{4}$$

The quotient in (4) is called the **cycle mean** of the cycle C. That is, the cycle mean of C is the sum of the execution times of all vertices on C divided by the path delay of C. The entire quantity on the RHS of (4) is called the "maximum cycle mean" of the strongly connected IPC graph G. If the IPC graph contains more than one SCC, then different SCCs may have different asymptotic iteration periods, depending on their individual maximum cycle means. In such a case, the iteration period of the overall graph (and hence the self-timed schedule) is the *maximum* over the maximum cycle means of all the SCCs of G_{ipc}. This is because the execution of the schedule is constrained by the slowest component in the system. Henceforth, we will use the following definition for the maximum cycle mean.

Maximum cycle mean & Critical Cycle

Definition 2: The **maximum cycle mean** of an IPC graph G_{ipc}, denoted by λ_{max}, is the maximal cycle mean over all strongly connected components of G_{ipc}: That is,

$$\lambda_{max} = \max_{\text{cycle } C \text{ in } G} \left\{ \frac{\sum_{v \text{ is on } C} t(v)}{Delay(C)} \right\}.$$

A fundamental cycle in G_{ipc} whose cycle mean is equal to λ_{max} is called a **critical cycle** of G_{ipc}. Thus the throughput of the system of processors executing a particular self-timed schedule is equal to the corresponding $\frac{1}{\lambda_{max}}$ value.

Scheduling and graph analysis

- Edges represent buffers
- Edges not in a strongly connected component are not bounded.
- Simpler protocols can be used on bounded edges.
- An edge is redundant if another path between the source/sink pair has a longer delay.

In dataflow semantics, the edges between actors represent infinite buffers. Accordingly, the edges of the IPC graph are potentially buffers of infinite size. However, from Lemma 2, the number of tokens on each feedback edge (an edge that belongs to an SCC, and hence to some cycle) during the execution of the IPC graph is bounded above by a constant. We will call this constant the **self-timed buffer bound** of that edge, and for a feedback edge e we will represent this bound by $B_{fb}(e)$. Lemma 2 yields the following self-timed buffer bound:

$$B_{fb}(e) = min(\{Delay(C) | C \text{ is a cycle that contains } e\}) \tag{5}$$

Bounding Buffer Synchronization (BBS)

Feedforward edges have no such bound on buffer size; therefore for practical implementations we need to *impose* a bound on the sizes of these edges. For example, Fig. 2(a) shows an IPC graph where the IPC edge (A, B) could be unbounded when the execution time of A is less than that of B, for example. In practice, we need to bound the buffer size of such an edge; we will denote such an "imposed" bound for a feedforward edge e by $B_{ff}(e)$. Since the effect of placing such a restriction includes "artificially" constraining $src(e)$ from getting more than $B_{ff}(e)$ invocations ahead of $snk(e)$, its effect on the estimated throughput can be modelled by adding the reverse edge $d_m(snk(e), src(e))$, where $m = B_{ff}(e) - delay(e)$, to G_{ipc} (grey edge in Fig. 2(b)). Since adding this edge introduces a new cycle in G_{ipc}, it may reduce the estimated throughput; to prevent such a reduction, $B_{ff}(e)$ must be chosen large enough so that the maximum cycle mean remains unchanged upon adding $d_m(snk(e), src(e))$.

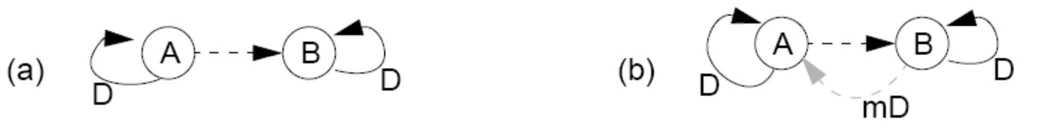


Fig. 2. An IPC graph with a feedforward edge: (a). original graph (b). imposing bounded buffers.

Deriving a strongly connected synchronization graph

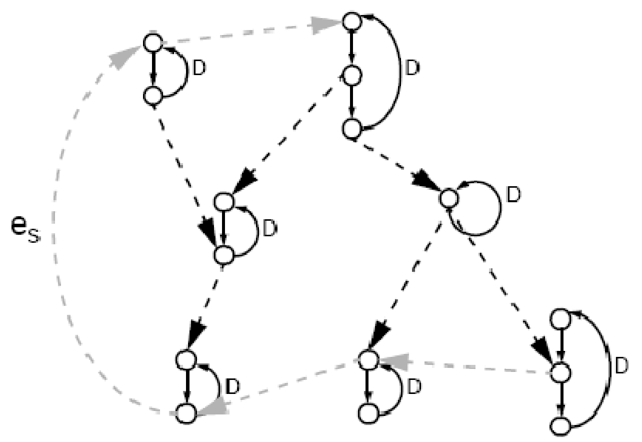


Figure 14. An illustration of a possible solution obtained by algorithm *Convert-to-SC-graph*.

Determine delays

Fig. 9 illustrates a solution obtained from *DetermineDelays*. Here we assume that $t(v) = 1$, for each vertex v, and we assume that the set of IPC edges is $\{e_a, e_b\}$. The grey dashed edges are the edges added by *Convert-to-SC-graph*. We see that λ_{max} is determined by the cycle in the sink SCC of the original graph; inspection of this cycle yields $\lambda_{max} = 4$. Also, the set W_0 — the set of fundamental cycles that contain e_0, and do not contain e_1 — consists of a single cycle c_0 that contains three edges. By

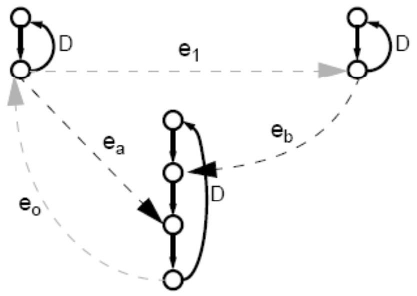


Fig. 9. An example used to illustrate a solution obtained by algorithm *DetermineDelays*.

Determine delays

- We need to add delays to the edges, corresponding to buffer memory,
  - that ensure the system will not deadlock
  - That we can minimize the sum of the buffer bounds over all the IPC edges.
- We can use the added edges to help us determine these delays.
  - The added edges can be divided into disjoint sets that help organize the graph.
- We can determine the minimum delay on each edge that ensures that the graph's cycle mean is not exceeded.

Complete Algorithm

```
Function SynchronizationOptimize
Input: A DFG $G$ and a self-timed schedule for this DFG.
Output: $G_{ipc}$, $G_s$, and $\{ B_{fb}(e) \mid e$ is an IPC edge in $G_{ipc} \}$.

1. Extract $G_{ipc}$ from $G$ and the given parallel schedule (which specifies actor assignment to
processors and the order in which each actor executes on a processor)

2. Set $G_s = G_{ipc}$                          /* Each IPC edge is also a synchronization
                                                 edge to begin with */

3. $G_s = RemoveRedundantSynchs(G_s)$

4. $G_s = Resynchronize(G_s)$

5. $G_s = Convert\text{-}to\text{-}SC\text{-}graph(G_s)$

6. $G_s = DetermineDelays(G_s)$

/* Remove any synchronization edges that have become redundant as a result of the appli-
cation of Convert-to-SC-graph. */
7. $G_s = RemoveRedundantSynchs(G_s)$

8. Calculate buffer sizes $B_{fb}(e)$ for each IPC edge $e$ in $G_{ipc}$. (to be used for implementing
the BBS protocol)
        a) Compute $\rho_{G_s}(src(e), snk(e))$, the path delay of a minimum-delay
        path in $G_s$ directed from $src(e)$ to $snk(e)$
        b) Set $B_{fb}(e) = \rho_{G_s}(src(e), snk(e)) + delay(e)$
```

Conclusions

We have addressed the problem of minimizing synchronization overhead when implementing self-timed, iterative dataflow programs. We have introduced a graph-theoretic analysis framework that allows us to determine the effects on throughput and buffer sizes of modifying the points in the target program at which synchronization functions are carried out, and we have used this framework to extend an existing technique — removal of redundant synchronization edges — for noniterative programs to the iterative case, and to develop two new methods for reducing synchronization overhead — resynchronization and the conversion of the synchronization graph into a strongly connected graph. Finally, we have shown how our techniques can be combined, and how the result can be post processed to yield a format from which IPC code can easily be generated.

Data dependency + Rate-monotonic

- Assume that there is a set of processes with data dependencies between them; in general, they can form one or more subtasks.
- Also assume that each CPU schedules processes rate-monotonic scheduling.
- The combination of data dependencies and rate-monotonic scheduling makes the problem more challenging, although tractable.

Bounds of the response times for a set of independent processes (Lehoczky)

- Suppose P1, P2, … are a set of priority-ordered processes allocated on the same CPU.
- For a process Pi, its minimum period is pi, and its longest computation time is ci.
- Let the worst-case response time form a request of Pi to its finish be wi
- Lehoczky showed that wi is the smallest nonnegative root of the equation

$$x=g(x)=ci + \sum_{j=1}^{i-1} cj \bullet \lceil x/pj \rceil$$

- It can be solved with a fixed-point iteration technique.

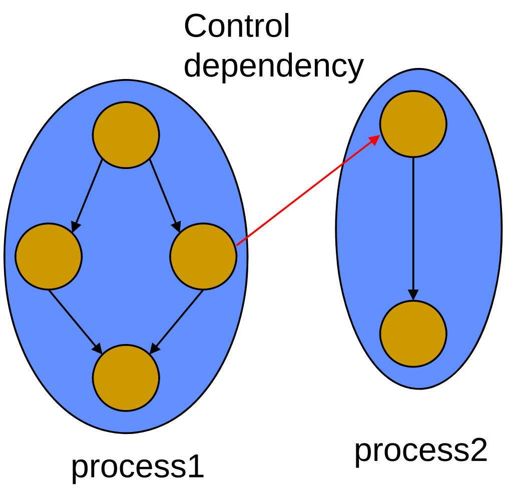
Fixed-point iteration technique and an example

- The fixed-point iteration technique
  (1) $w = \lceil c_i / (1 - \sum_{j=1}^{i-1} c_j/p_j) \rceil$
  (2) while (w <g(w)) w = g(w)

- (example) Suppose p1=5, c1=1, p2=37, c2=3, p3=51, c3=16, p4=134, c4=42.
- Fixed-point iteration tells us that we only need four steps to know that w4= 128; the x values during iterations are 104, 120, 126, and 128.

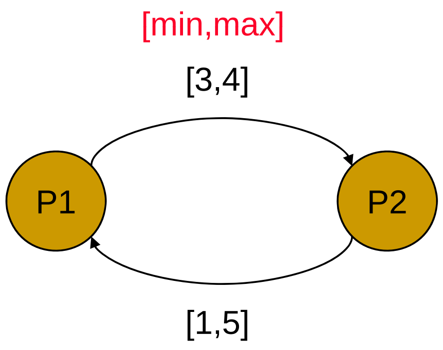
Rate analysis (Gupta)

- Goal: identify execution rates at which processes can run while satisfying the min-max bounds of delays
- Model includes multiple processes with control dependencies.
  - A CDFG-style model within each process.



Process model

- Edges are labeled with (min,max) delays from activation signal to start of execution.
- Process starts executing after all its enables signals have been ready.

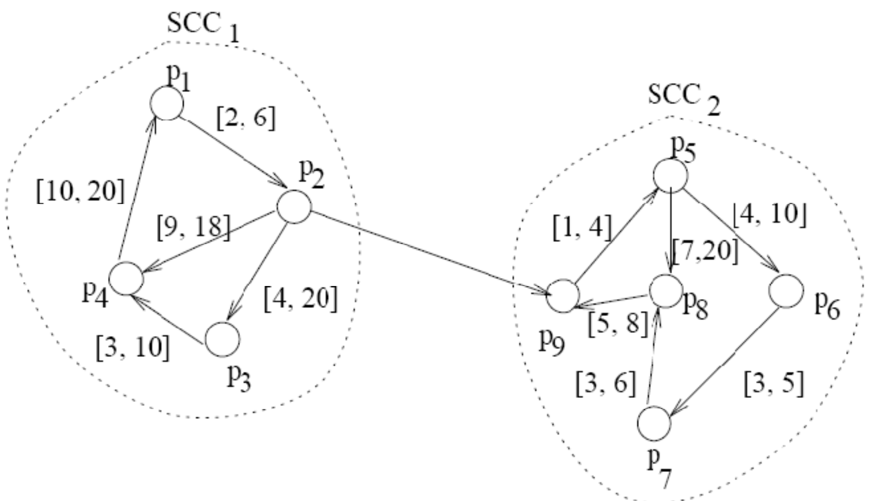


Rate analysis

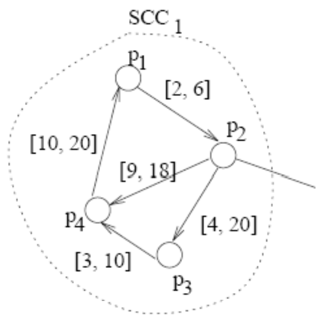
- Delay around a cycle in the graph is $\Sigma\ \delta_i$.
- Maximum mean cycle delay is λ.
- In a strongly connected graph all nodes execute at the same rate λ.
- Given a producer and consumer, bounds on rates of consumer is:

  $[\ \min\{r_l(P),r_l(C)\},\ \min\{r_u(P),r_u(C)\}\ ]$

Rate analysis example



Rate analysis example



For SCC_1:

For computing r_l, set all the edge delays to their upper bounds.

$$\begin{aligned}\lambda_l &= \text{maximum mean delay cycle in } SCC_1\\ &= \max\left\{\frac{20+18+6}{3}, \frac{6+20+10+20}{4}\right\}\\ &= \max\left\{\frac{44}{3}, 14\right\}\\ &= 14.67\end{aligned}$$

The critical cycle is $(p_1 \rightarrow p_2 \rightarrow p_4 \rightarrow p_1)$ and

$$r_l = (14.67)^{-1} = 0.068$$

For computing r_u, we use the lower bounds on the edge delays.

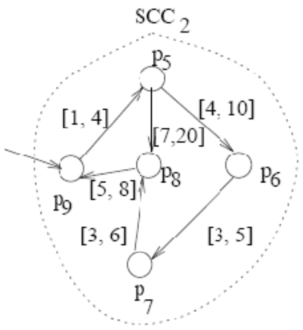
$$\begin{aligned}\lambda_u &= \text{maximum mean delay cycle in } SCC_1\\ &= \max\left\{\frac{10+9+2}{3}, \frac{2+4+3+10}{4}\right\}\\ &= \max\{7, 4.75\}\\ &= 7\end{aligned}$$

Thus,

$$r_u = (7)^{-1} = 0.142$$

Hence, the rate interval for SCC_1 is $[0.068, 0.142]$.

Rate analysis example



For SCC_2:

$$r_l = \left[\max\left\{\frac{20+8+4}{3}, \frac{10+5+6+8+4}{5}\right\}\right]^{-1} = 0.094$$

and

$$r_u = \left[\max\left\{\frac{7+5+1}{3}, \frac{4+3+3+5+1}{5}\right\}\right]^{-1} = 0.231$$

So, the rate interval for SCC_2 is $[0.094, 0.231]$. Notice that the rate intervals of the two SCCs overlap. The rate analysis was carried out assuming that the two SCCs are completely disjoint. The rates in SCC_1 are not affected by the edge from SCC_1 to SCC_2, however, the rate interval for SCC_2 needs to take into consideration the rate interval of the "producer" SCC. In fact, the actual rate interval for SCC_2 is

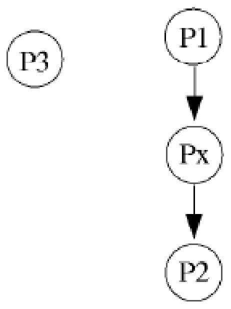
$$\begin{aligned} & [\min\{0.094, 0.068\}, \min\{0.231, 0.142\}] \\ = & [0.068, 0.142] \end{aligned}$$

Thus, the rate interval of SCC_2 is the same as that of SCC_1. Notice that for the sake of clarity of exposition, we have computed the maximum mean cycle delay using explicit enumeration of the cycles in this example. In our implementation, instead of explicit cycle enumeration, we use Karp's characterization of maximum mean cycle delay to compute the rates. □

Distributed system performance

- Performance analysis using longest-path algorithms don't work under preemptive scheduling.
- Several algorithms unroll the schedule to the length of the least common multiple of the periods:
  - produces a very long schedule;
  - doesn't work for non-fixed periods.
- Schedules based on upper bounds may give inaccurate results.
- Simulation does not provide guarantees.

Using worst case delay in unrolled schedules

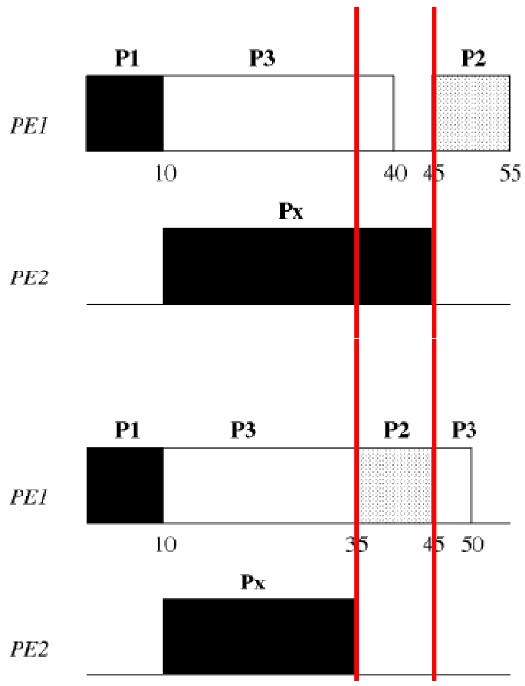


process	*period*	*computation time*	*allocation*	*deadline*
P_1	100	10	PE1	
P_2	100	10	PE1	
P_3	150	30	PE1	45
P_x	100	[25, 35]	PE2	

process characteristics

Changing the computation time for Px changes the response time of P3
Even though they run on different processors.

Using worst case delay in unrolled schedules



When Px computation time is 35, the response time of P3 is 40.

When Px computation time is 25, the response time of P3 is 50.

P3 deadline: 45 !

temporal behavior

A task graph and its implementation

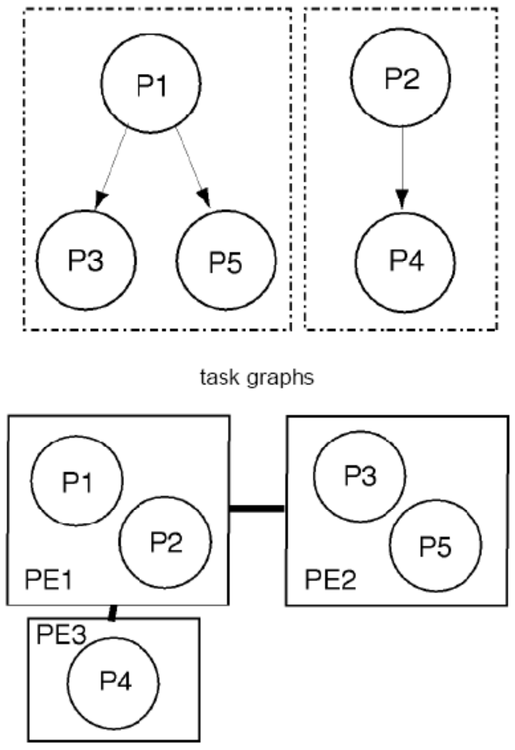
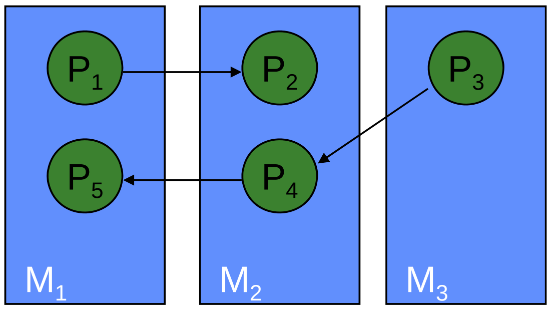


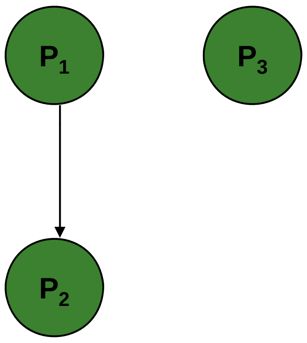
Fig. 2. A task graph and its implementation.

Preemptive execution hurts

- Two subtasks are divided into three processors
- Worst combination of events for P_5's response time:
  - P_2 of higher priority
  - P_2 initiated before P_4
  - causes P_5 to wait for finishing P_2 and P_3.
- Independent tasks can interfere—can't use longest path algorithms.



Data dependencies help

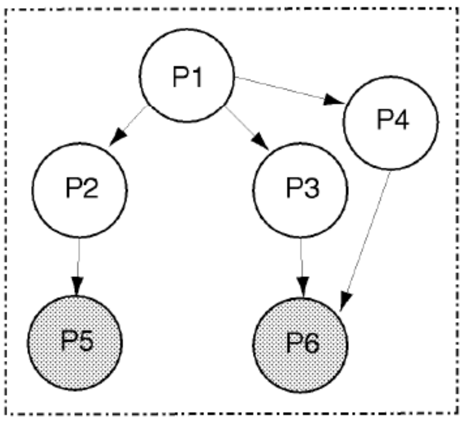


- P_3 cannot preempt both P_1 and P_2.
- P_1 cannot preempt P_2.
- If we ignore data dependencies, the worst response times are
  - 35 for P2
  - 45 for P3
- But the worst case total delay along the path from P2 to P3 is 45 instead of 80 (35+45)

	Period	Computation time
P1	80	15
P2	50	20
P3	50	10

Separation analysis

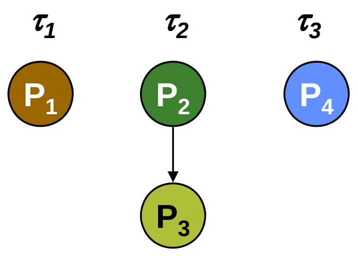
- P5, P6 : on the same PE
- Cases 1&2: P5 will not preempt P6 because they are separated
- Cases 3&4: P5 will preempt P6.



	case 1	case 2	case 3	case 4
P1	[12,15] →			
P2	[5,6]	[20,25] →		
P3	[20,25]	[5,6]	[5,12]	[5,6]
P4	[8,10] →			[30,35]
P5	[10,10] →			
P6	[10,10] →			

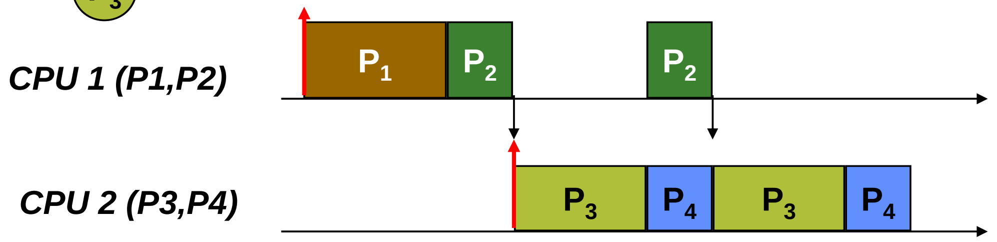
Fig. 7. Several different schedules for a task showing different combinations of execution time overlaps.

Period shifting example



task	period
τ_1	150
τ_2	70
τ_3	110

process	CPU time
P_1	30
P_2	10
P_3	30
P_4	20



- P_2 delayed 30 on CPU 1 due to P_1
- data dependency delays P_3 20 more
- priority of P_3 delays P_4 by preemption.
- Worst case delay of task 3 is 30+30+20=80, but 30+20=50

Period shifting

- The delay for the preprocessors may vary from period to period, making the request period of a process different form the period of the task.