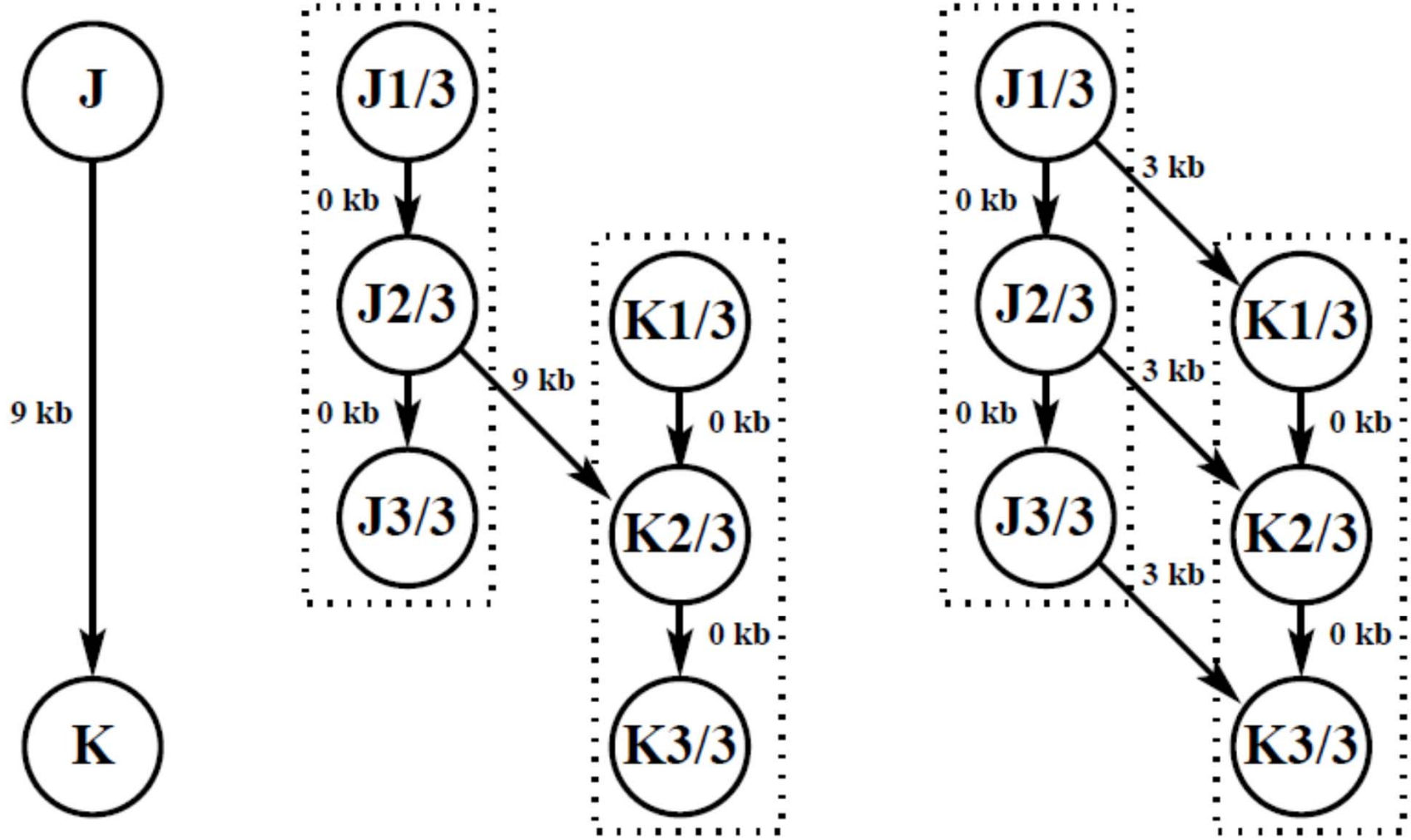
Distributed Embedded Systems With Real time Requirements

A distributed system is a collection of individual computing devices that can communicate with each other and execute a complex task with deadlines together and efficiently

In many applications distributed systems are necessary because the devices may be physically separated.

If the deadlines for processing data are short it may be more cost effective to put the PE’s where the data are located rather than build a high bandwidth network to carry data to a distant PE.

Possible Solutions

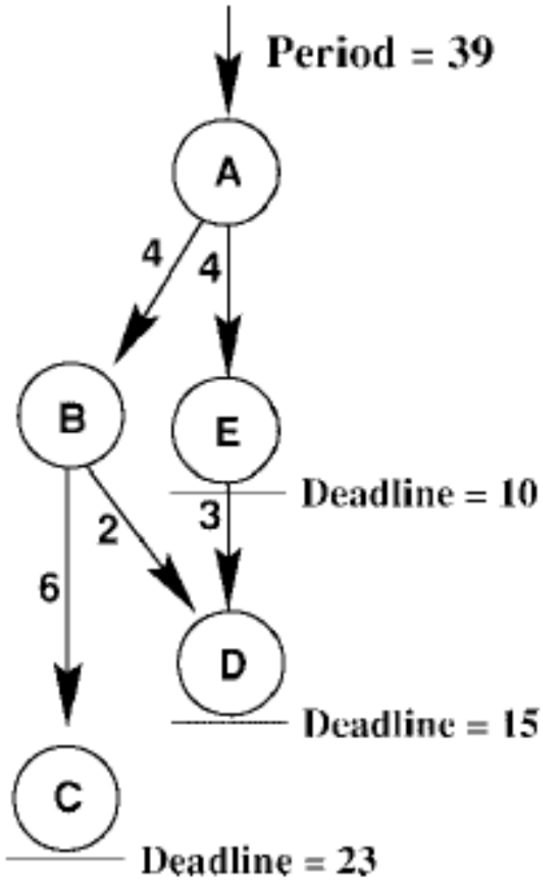


a) conventional

b) pre– and post– computation

c) streaming

Task Graph

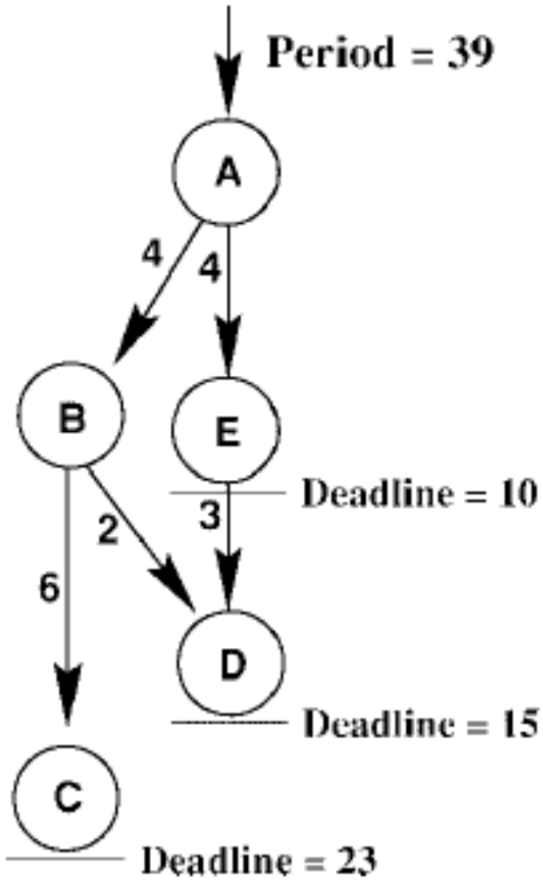


Task Graph: Task graphs specify some of the requirements a designer places upon an embedded system.

Is a directed acyclic graph in which each node is associated with a task and each edge is associated with a scalar describing the amount of data that must be transferred between the two connected tasks.

Each task may only begin executing after all of its data dependencies have been satisfied.

Task Graph



The *period* of a task graph is the amount of time between the earliest start times of its consecutive executions.

A node with no outgoing edges is called a *sink* node.

A *deadline,* the time by which the task associated with the node must complete its execution, exists for every sink node.

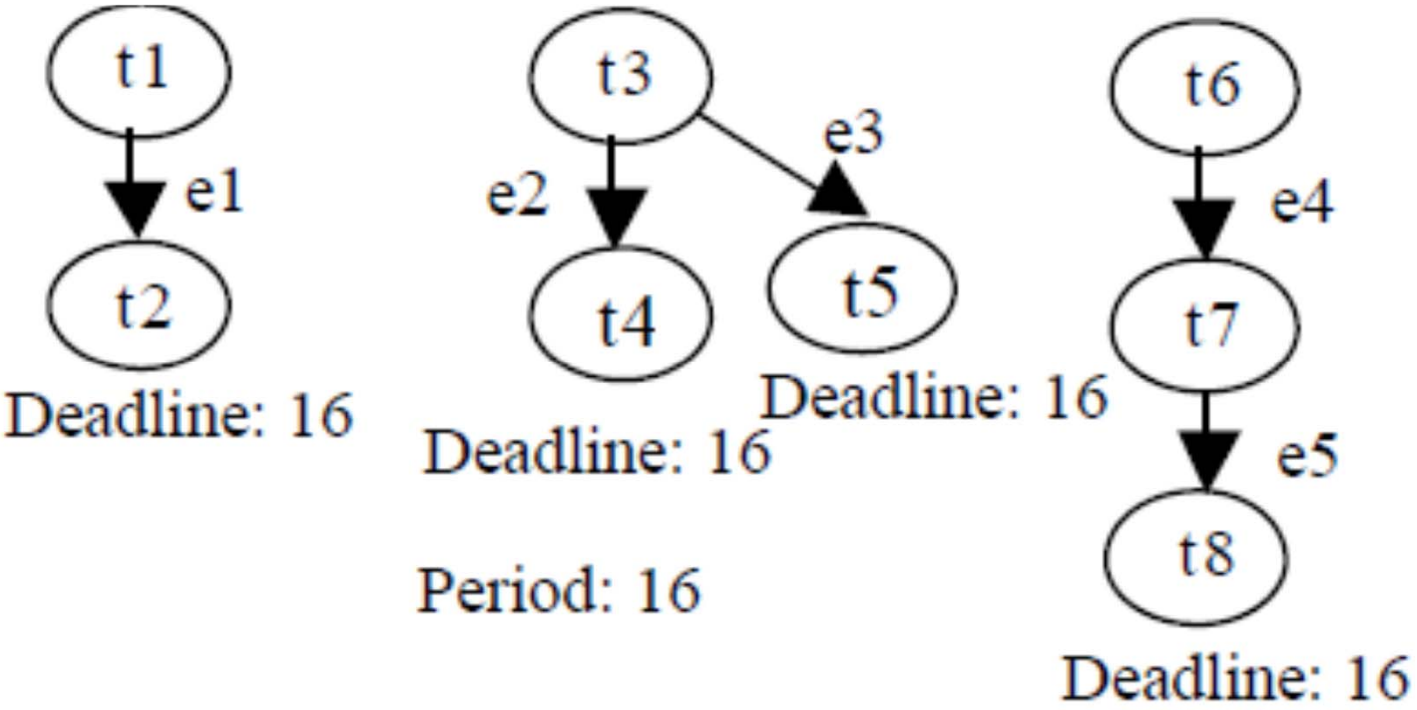
Other nodes may also have deadlines associated with them.

The deadline of a task graph is the maximum of all the deadlines specified in it.

An embedded system specification may contain multiple task graphs, each of which may have a different period.

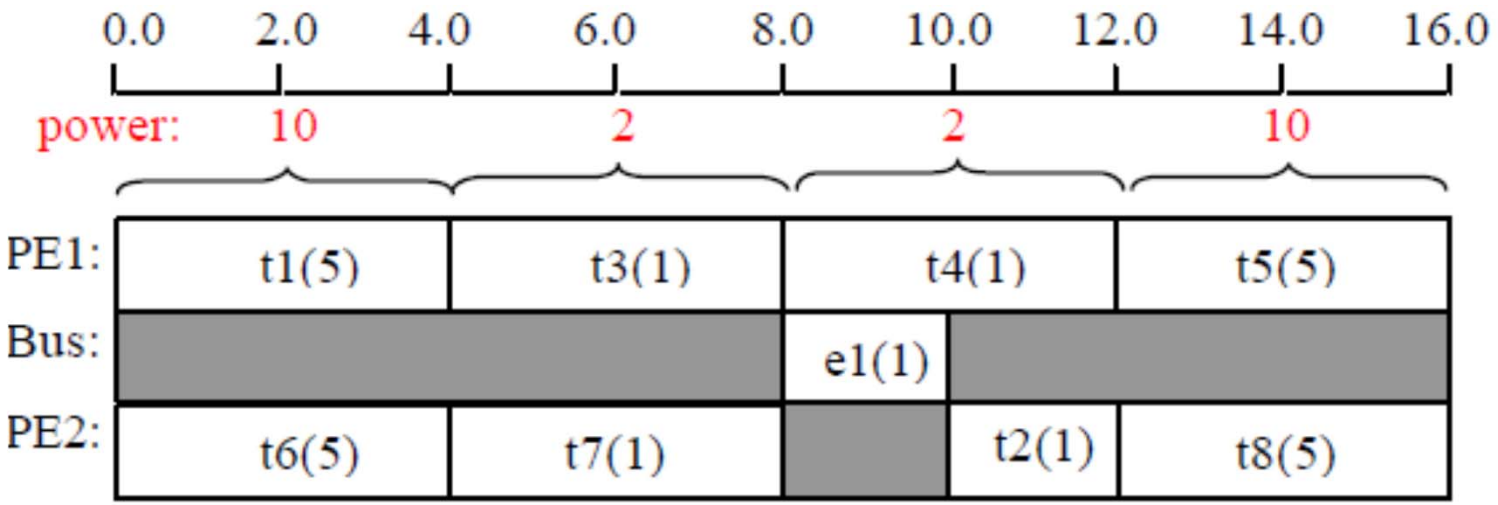
Scheduling on Two processors: Task Graph

Execution Time for t2 is 2 seconds
All other jobs take 4 seconds. Bus time for e1 is 2 seconds.
All other bus times are zero



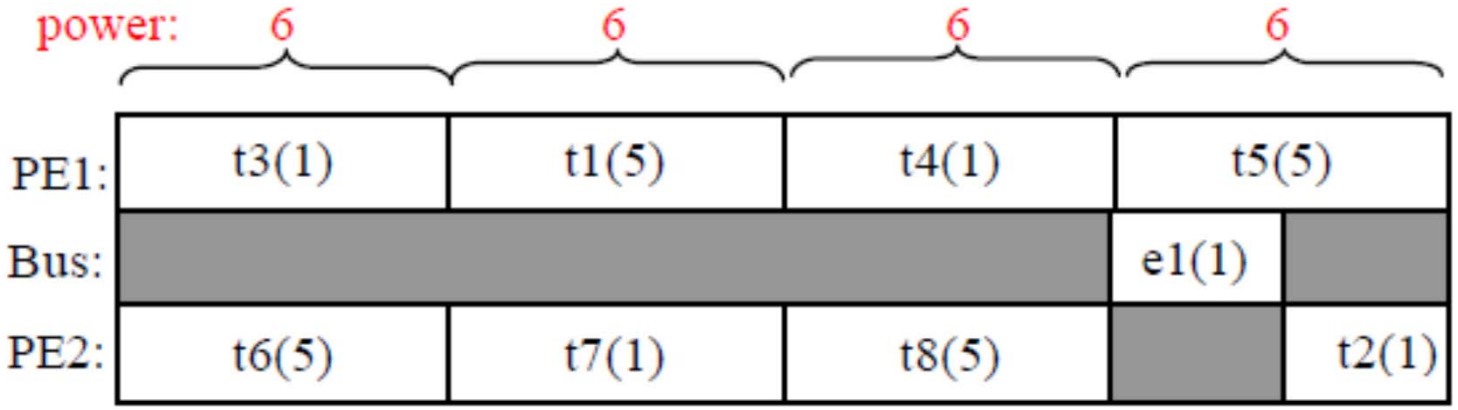
Valid Scheduling on Two processors: Task Graph for Example 1

Execution Time for t2 is 2 seconds
All other jobs take 4 seconds. Bus time for e1 is 2 seconds.
All other bus times are zero



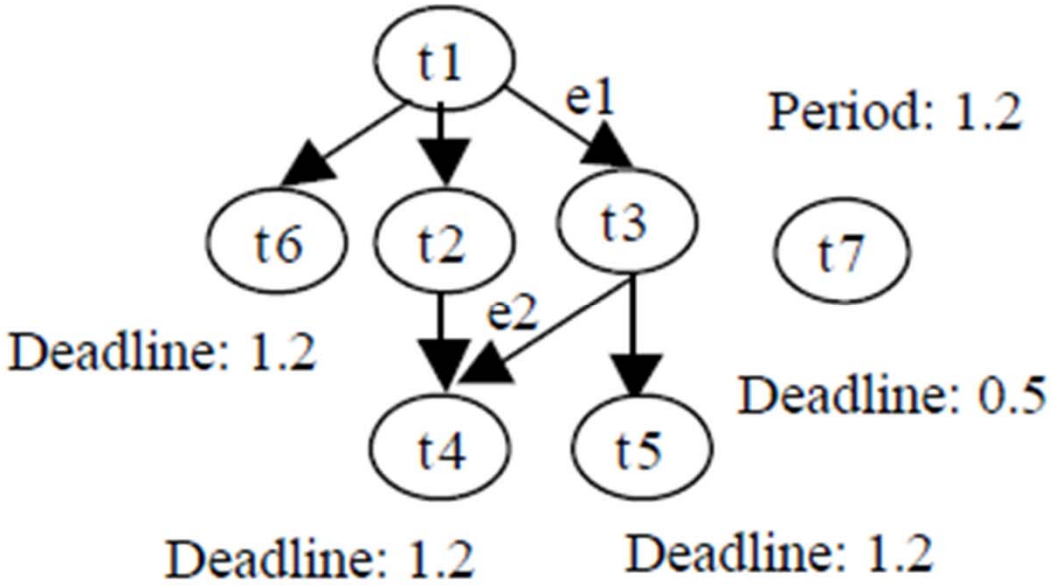
Power Efficient Scheduling on Two processors: Task Graph for Example 1

Execution Time for t2 is 2 seconds
All other jobs take 4 seconds. Bus time for e1 is 2 seconds.
All other bus times are zero



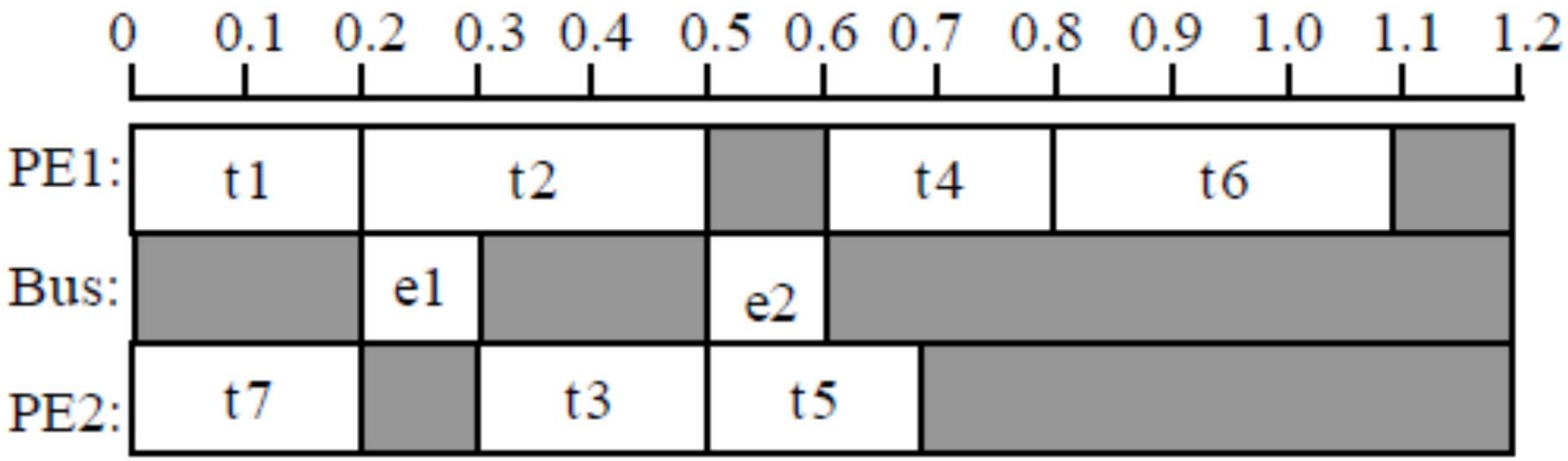
Scheduling on Two processors: Task Graph for Example 2

Execution Time for t1,t3,t4,t5,t7 .2 seconds, for t2 and t6 it is .3 seconds
Bus time for e1 and e2 is .1 seconds.
Average Power consumed per each task is 1 unit, for bus communication is .2 units



A Scheduling on Two processors: Task Graph for Example 2

Execution Time for t1,t3,t4,t5,t7 .2 seconds, for t2 and t6 it is .3 seconds
Bus time for e1 and e2 is .1 seconds.
Average Power consumed per each task is 1 unit, for bus communication is .2 units



A Power eff. Scheduling on Two processors: Task Graph for Example 2

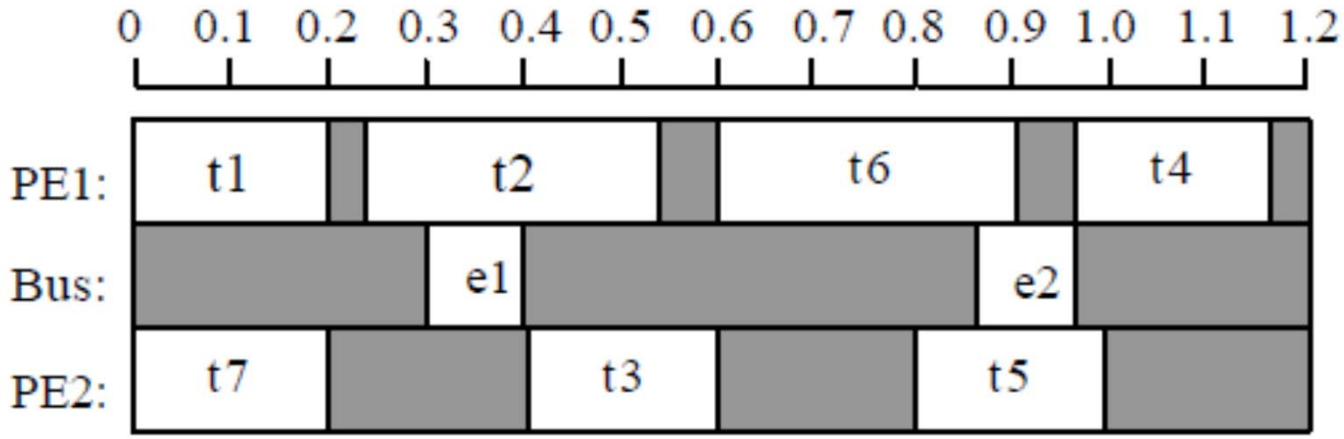
Execution Time for t1,t3,t4,t5,t7 .2 seconds, for t2 and t6 it is .3 seconds
Bus time for e1 and e2 is .1 seconds.
Average Power consumed per each task is 1 unit, for bus communication is .2 units

t2 is scheduled at time instant 0.2. Since it can finish as late as time instant 0.6, the speed of PE1 can be scaled down by a ratio of (0.6 – 0.2) / 0.3 for *t2*. Correspondingly, the supply voltage can be scaled down from 3.3V to 2.8V, extending the actual running length of *t2* from 0.3 to 0.4.

The working voltages for task *t1*, *t2*, *t3*, *t4*, *t5*, *t6* and *t7* are 3.3, 2.8, 3,3, 3.3, 1.8, 2.8, and 2.7V, respectively.

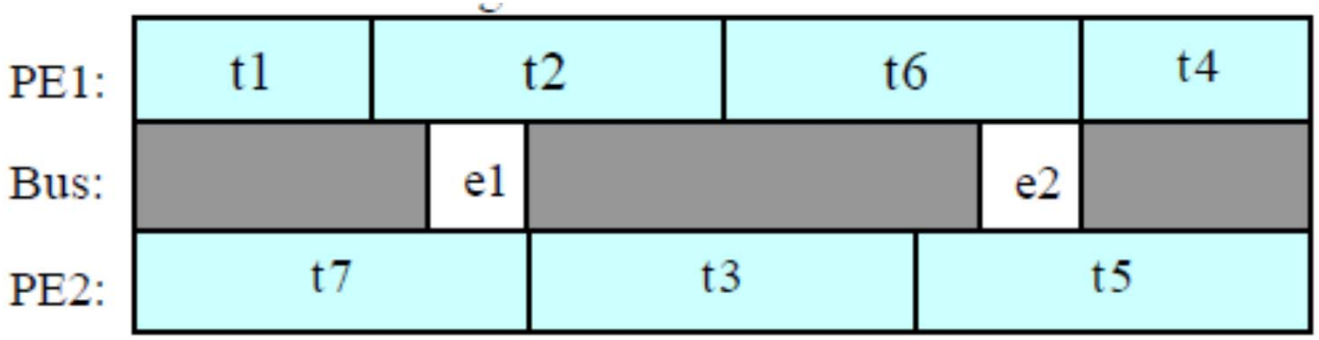
Another Scheduling on Two processors: Task Graph for Example 2

Execution Time for t1,t3,t4,t5,t7 .2 seconds, for t2 and t6 it is .3 seconds
Bus time for e1 and e2 is .1 seconds.
Average Power consumed per each task is 1 unit, for bus communication is .2 units



Another Scheduling on Two processors: Task Graph for Example 2

Execution Time for t1,t3,t4,t5,t7 .2 seconds, for t2 and t6 it is .3 seconds
Bus time for e1 and e2 is .1 seconds.
Average Power consumed per each task is 1 unit, for bus communication is .2 units



Distributed Embedded System Design

- Process of concurrently defining the hardware and software portions of an embedded system while considering dependencies between the two.

- Designers rely on their experience with past systems when estimating the resource requirements of a new system.

- Since ad hoc design exploration is time consuming, an engineer typically selects a conservative architecture after little experimentation, resulting in an unnecessarily expensive system.

Distributed Embedded System Design : New Approach

• Research in the area of hardware–software codesign has focused on easing the process of design exploration.

•Automating this process falls within the more specialized realm of cosynthesis.

•

•Given an embedded system specification, a cosynthesis system determines the hardware and software processing elements (PE's) needed as well as the communication links to be used.

• The system assigns each task to a PE and determines the PE's to which each link is connected.

•A schedule is provided for each PE and communication link such that all real-time constraints are met.

•Cosynthesis systems generate feasible, low-cost architecture descriptions without designer intervention.

Distributed Embedded System Design : New Approach : Requirements

- Systems are composed of multiple general-purpose processors and ASIC’s and buses.

- Power consumption is often a concern during the design of embedded systems. It is important to reduce the average power consumption of such systems, thereby increasing battery lifespan.

- Early work in low power electronics focused on changes to fabrication technology and logic design, it has been shown that larger gains can be obtained by considering power during the earlier phases of the design process

Distributed Embedded System Design : New Approach : Requirements

Communication links consume power as well as time. A cosynthesis system that targets low-power applications must take both PE and communication link power requirements into account.

Many real distributed embedded systems are composed of numerous PE’s, and there are several types of communication links available for connecting them. A practical cosynthesis system must be capable of automatically generating a low-price, low-power, heterogeneous communication network.

Four Tasks to be considered by the Cosynthesis System

- *Allocation:* Determine the quantity of each type of PE and communication link to use.
- *Assignment:* Select a PE to execute each task upon. Choose a link to use for each communication event.
- *Scheduling:* Determine the time at which each task and communication event occurs.
- *Performance evaluation:* Compute the price, speed, and power consumption of the solution.

Intractability of Cosynthesis Problem

Optimal cosynthesis is an intractable problem.

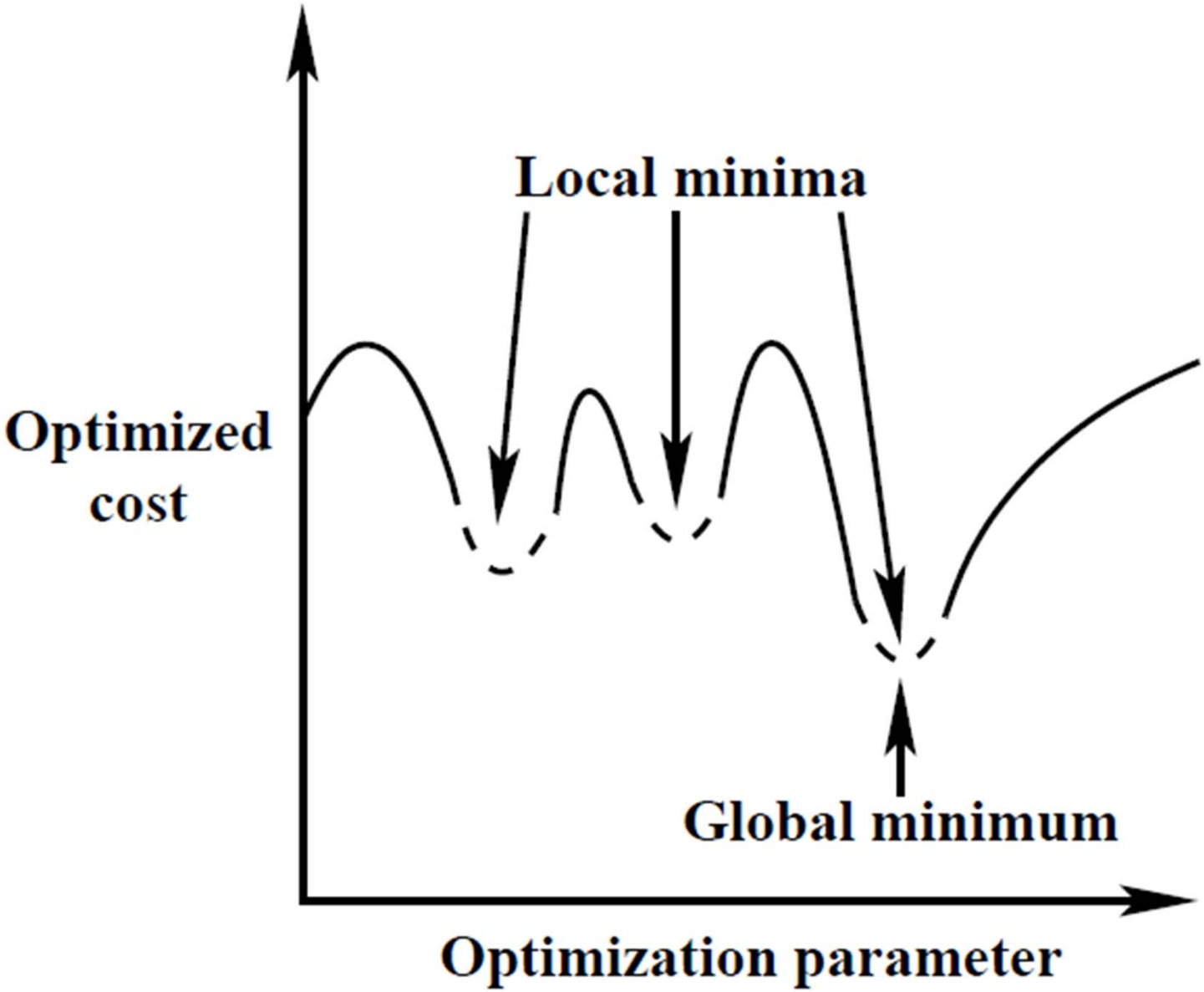
Allocation/assignment and scheduling are each known to be NP-complete for distributed systems.

Cosynthesis systems rely on optimal mixed integer linear programming and exhaustive exploration

- can only be applied to small instances of the cosynthesis problem.

Heuristics have seen some success with larger instances of the distributed system cosynthesis problem.

Local and Global Minima



Heuristics for Cosynthesis Problem

1. Iterative improvement algorithms start with a complete, but suboptimal, solution and make local changes to it while monitoring the solution's cost.

-- Algorithms in this class are prone to becoming trapped in local minima.

2. Constructive algorithms build a system by incrementally adding components. To be computationally tractable, a constructive algorithm must make changes with global impact while inspecting only the local effects of these changes. This often leads to an accumulation of suboptimal decisions, especially when large systems are constructed and can get trapped in local minima,

Probabilistic Algorithms for Cosynthesis Problem

Simulated Annealing algorithms: Simulated annealing algorithms are capable of escaping local minima of arbitrary depth. Simulated annealing algorithms have been successfully used to partition hardware–software systems.

Genetic algorithms: In general, genetic algorithms share simulated annealing algorithms' ability to escape local minima,

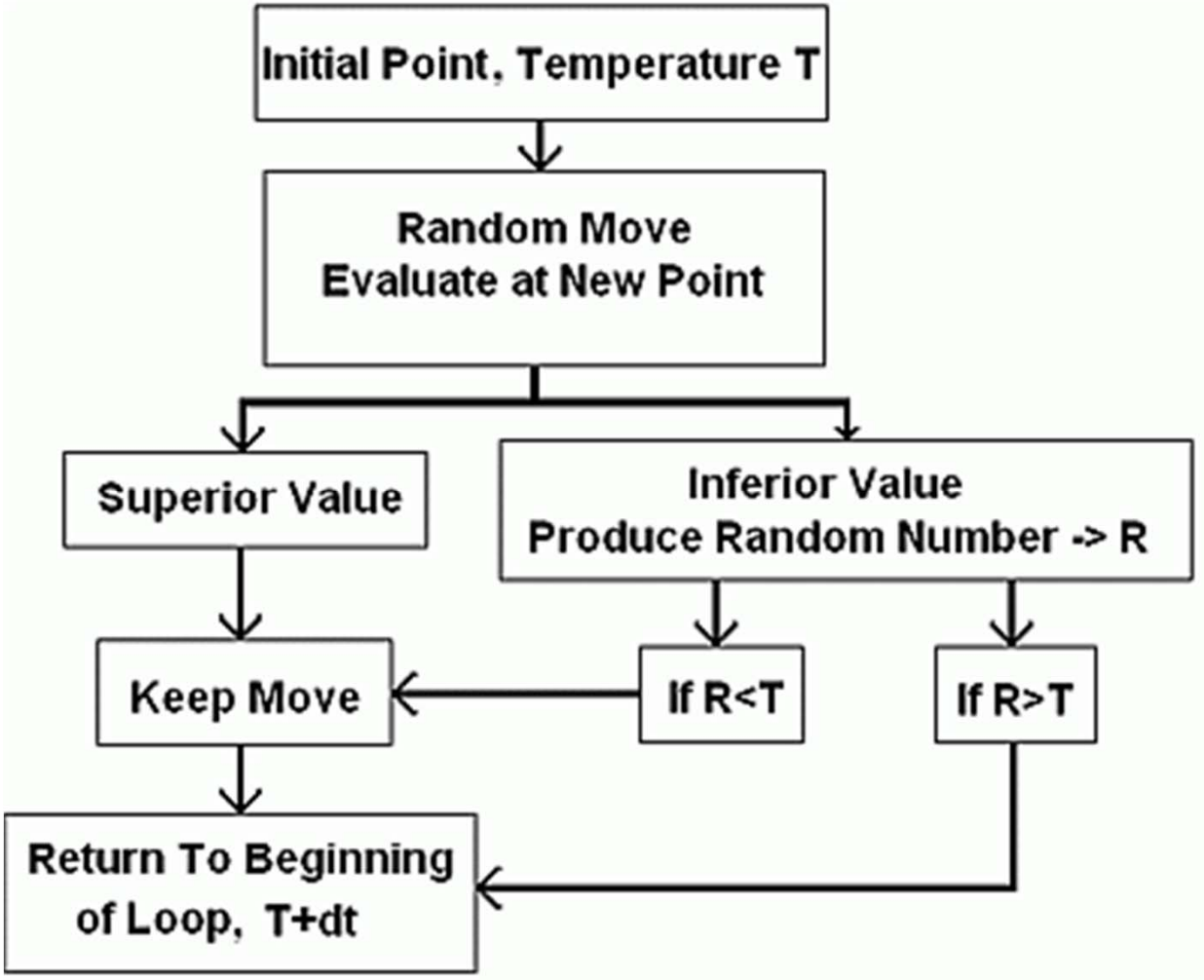
Genetic algorithms allow solutions to cooperatively share information with each other.

They are capable of true multi objective optimization, exploring the set of solutions that can only be improved in one way by being degraded in another

Simulated Annealing

Simulated Annealing (SA) is a multidimensional optimization method inspired by the metallurgical process of annealing. In metal, this is accomplished by heating a specimen and allowing the molecules to diffuse to more stable positions. As the temperature is slowly lowered the molecules settle to positions which result in lower internal stresses. The same general process is used in simulated annealing where a temperature value is assigned to the function, and the optimization occurs as that temperature drops.

Simulated Annealing: Flow Chart



Genetic Algorithm

Genetic algorithms are inspired by Darwin's theory about evolution. Solution to a problem solved by genetic algorithms is evolved.

Algorithm is started with a **set of solutions** (represented by **chromosomes**) called **population**. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (**offspring**) are selected according to their fitness - the more suitable they are the more chances they have to reproduce.

Genetic Algorithms

Genetic algorithms maintain a pool of solutions that evolve in parallel over time.

Genetic operators are applied to the solutions in the current pool to improve the solutions.

The lowest quality solutions are then removed from the pool.

A cost is a variable that a genetic algorithm attempts to minimize, e.g., price and power consumption.

Genetic algorithms excel at simultaneously optimizing multiple conflicting costs. They have the ability to escape local minima and communicate information among solutions.

Genetic Algorithm

Outline of the Basic Genetic Algorithm

1. **[Start]** Generate random population of *n* solutions (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness *f(x)* of each solution *x* in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
   a. **[Selection]** Select two parent solutions from a population according to their fitness (the better fitness, the bigger chance to be selected)
   b. **[Crossover]** With a crossover probability cross over the parent solutions to form a new solution. If no crossover was performed, offspring is an exact copy of parents.
   c. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in the solution).
   d. **[Accepting]** Place new offspring in a new population
4. **[Replace]** Use new generated population for a further run of algorithm
5. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in current population
6. **[Loop]** Go to step **2**

Genetic Algorithms : Three operators

1. Reproduction makes a copy of a solution.

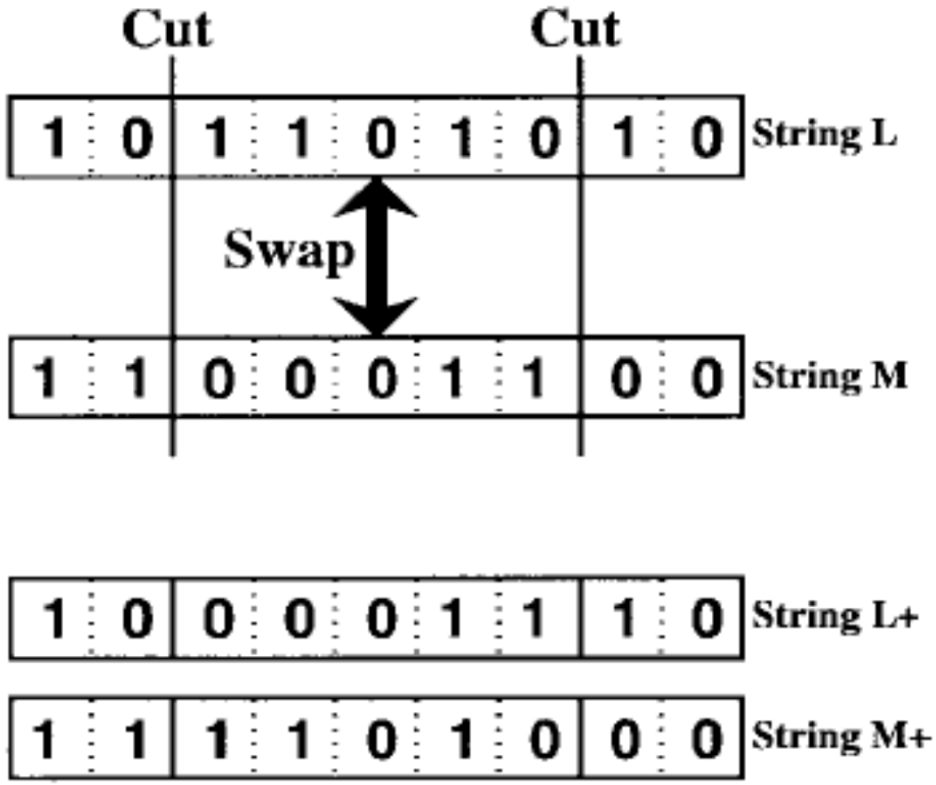
2. Mutation randomly changes part of a solution's description.

3. Crossover swaps portions of different solutions.

Some genetic algorithms are capable of varying the probability of allowing a solution to be replaced by one of lower quality.

.

Genetic Algorithms : Crossover operator



MOGAC: A Multiobjective Genetic Algorithm for HW/SW Cosynthesis of Distributed Embedded Systems

Abstract— In this paper, we present a hardware–software **cosynthesis system, called MOGAC, that partitions and schedules embedded system specifications consisting of multiple periodic task graphs. MOGAC synthesizes real-time heterogeneous distributed architectures using an adaptive multiobjective genetic algorithm that can escape local minima. Price and power consumption are optimized while hard real-time constraints are met. MOGAC places no limit on the number of hardware or software processing elements in the architectures it synthesizes.**

MOGAC: A Multiobjective Genetic Algorithm for HW/SW Cosynthesis of DES

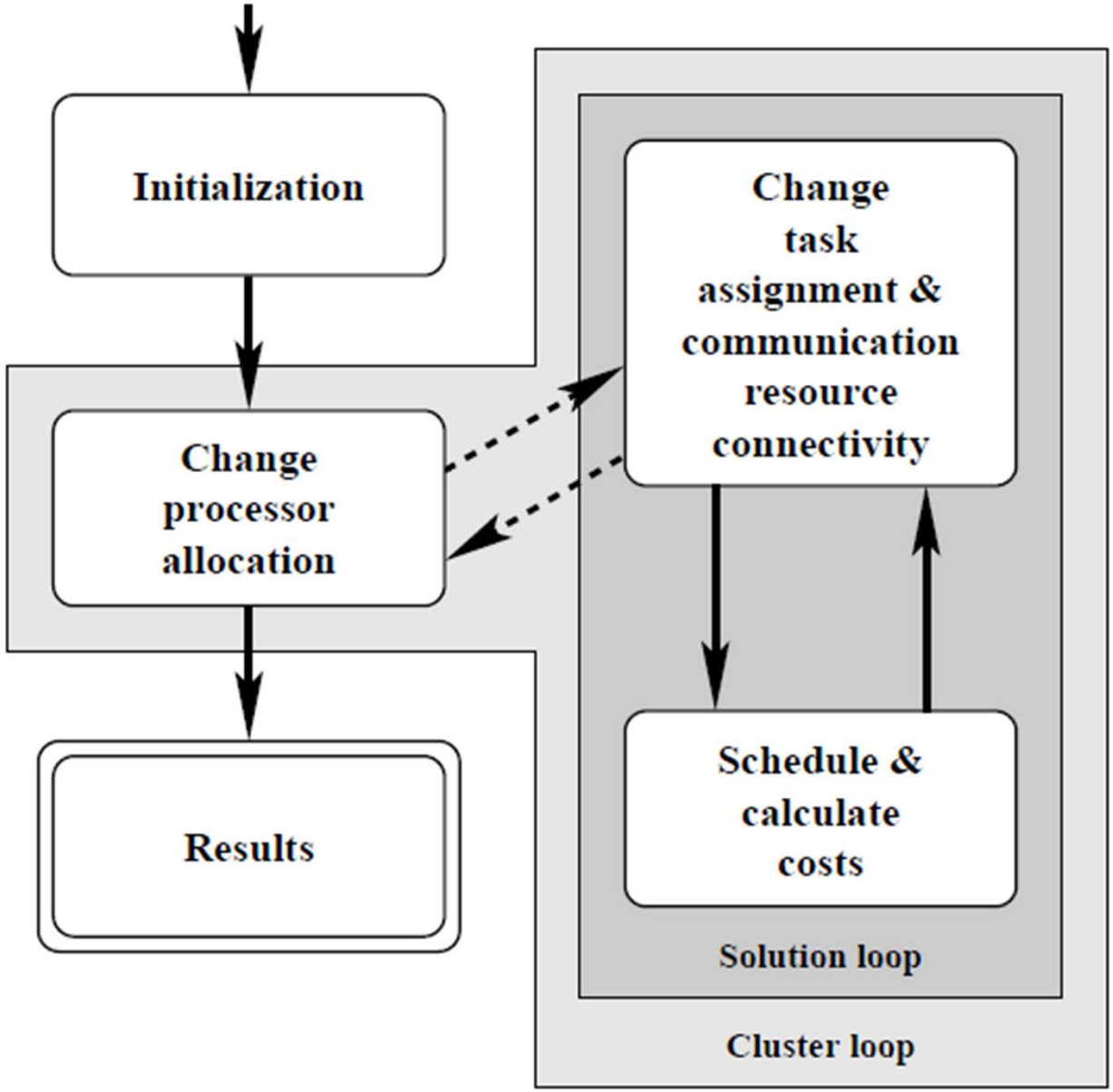
- Cosynthesis of low-power, real-time, multirate heterogeneous hardware–software distributed embedded systems.

- A multiobjective genetic algorithm, which allows exploration of optimal set of architectures instead of providing a designer with a single solution.

- Practically applied to a number of examples found in the literature. MOGAC has been shown to rapidly synthesize architectures with costs that are lower than or equal to those presented in previous work.

For large examples upon which comparisons with other systems are possible, MOGAC produces significantly lower cost solutions, requiring orders of magnitude less run time.

MOGAC's optimization infrastructure



Four Tasks to be considered by the Cosynthesis System

- *Allocation:* Determine the quantity of each type of PE and communication link to use.
- *Assignment:* Select a PE to execute each task upon. Choose a link to use for each communication event.
- *Scheduling:* Determine the time at which each task and communication event occurs.
- *Performance evaluation:* Compute the price, speed, and power consumption of the solution.

Relationship Between Tasks and Cores (PE)

A two-dimensional array indicating the relative worst-case execution time of each task on each core.

A two-dimensional array indicating the relative average power consumption of each task on each core.

A two-dimensional array indicating the peak power consumption of each task on each core.

More about Cores (PEs)

In MOGAC, cores do not have inherent prices. However, each core is assigned to an IC that does have a price.

The following variables are associated with ICs: price, device count, pins available, idle power consumption, peak power dissipation, speed, and power efficiency.

Solution representation

Each solution is represented by a collection of multidimensional data structures. The PE allocation is held in a one dimensional array of integers.

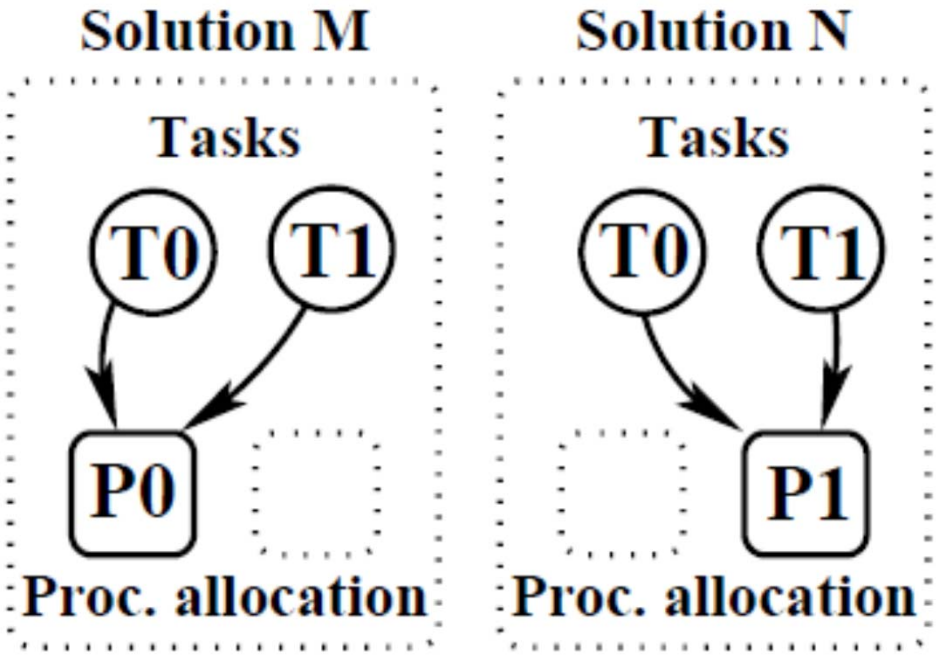
The offset into this array corresponds to PE type. The integer at each offset represents the number of PEs of that type in a solution. The communication resource allocation is represented by a similar one-dimensional array.

Task assignment is represented by a two-dimensional array in which the first dimension corresponds to the task graph that a task belongs to, and the second dimension corresponds to the task's index within the graph. Each entry in the array holds a one dimensional array with two entries.

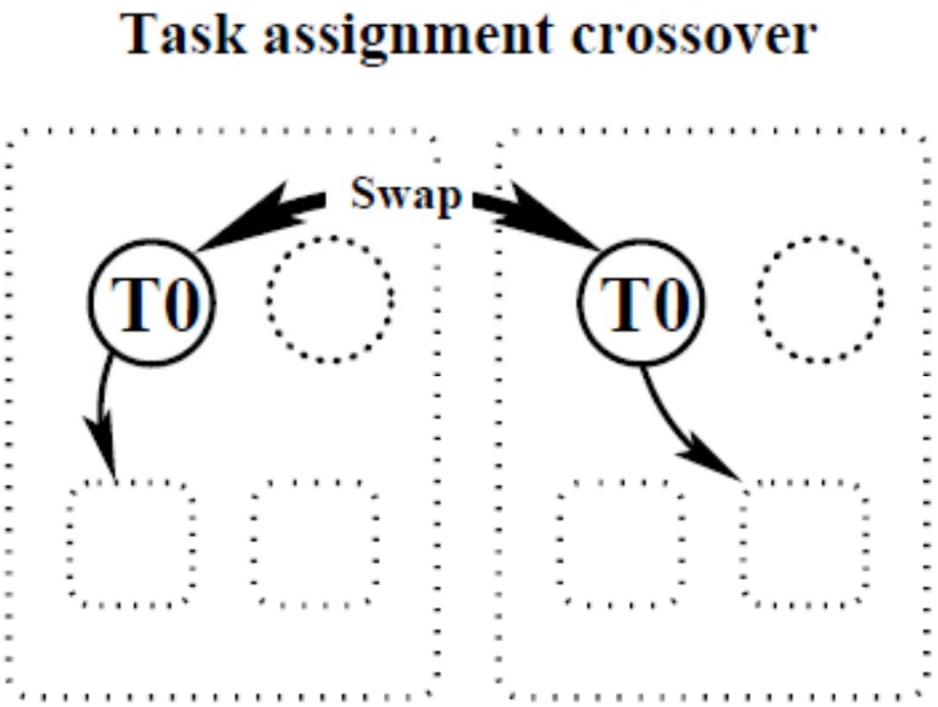
Clusters

Clusters of solutions are used to prevent crossover from producing *structurally incorrect* solutions, i.e., solutions that are physically impossible.

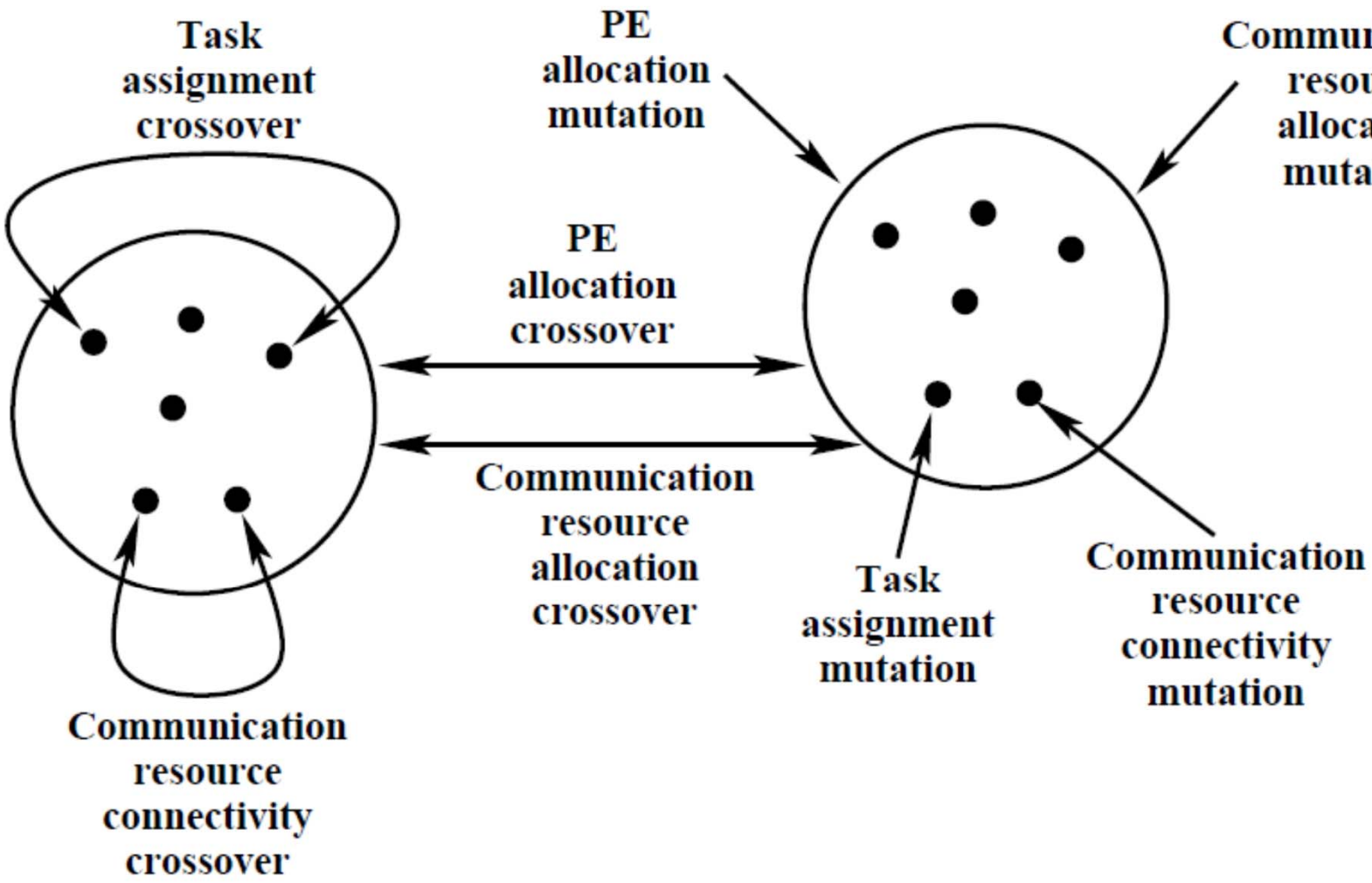
Clusters


Solution M
Tasks
T0
T1
P0
Proc. allocation
Solution N
Tasks
T0
T1
P1
Proc. allocation

Clusters



Solution Clusters



MOGAC Software Features

MOGAC is a prototype consisting of approximately 18 000 lines of C and Bison code. Our results were obtained on a 200-MHz Pentium Pro system with 96 MB of main memory running the Linux operating system.

MOGAC Software Inputs

MOGAC's input consists of two ASCII files. The first file specifies the attributes of each PE, IC, and link type that may be used to implement an architecture. In addition, this file specifies the relationships between PE's and tasks, i.e., for each PE, it contains arrays specifying the worst case execution times, average power consumptions, and peak power consumptions of each task on that PE.

MOGAC Software Inputs

The second file specifies the topologies, periods, deadlines, tasks, and communication flows associated with all the task graphs composing the system specification.

MOGAC Software Output

MOGAC runs without designer intervention and, upon halting, outputs one or more solutions. Each solution is a system architecture consisting of a price, power consumption, PE allocation, IC allocation, link allocation, core assignments, task assignments, link connectivities, task schedules for each PE, and communication event schedules for each link.

MOGAC Performance

TABLE I
HOU'S EXAMPLES

Example	No. of Tasks	Yen's system		COSYN		MOGAC		
		Price	CPU time (s)	Price	CPU time (s)	Price	CPU time (s)	Tuned time (s)
Hou 1 & 2 (unclustered)	20	170	10,205	N. A.	N. A.	170	5.7	2.8
Hou 3 & 4 (unclustered)	20	210	11,550	N. A.	N. A.	170	8.0	1.6
Hou 1 & 2 (clustered)	8	170	16.0	170	5.1	170	5.1	0.7
Hou 3 & 4 (clustered)	6	170	3.3	N. A.	N. A.	170	2.2	0.6

References

1. Battery-aware Static Scheduling for Distributed Real-time Embedded Systems Jiong Luo and Niraj K. Jha Department of Electrical Engineering Princeton University, Princeton, NJ, 08544 {jiongluo, jha}@ee.princeton.edu

2. MOGAC: A Multiobjective Genetic Algorithm for Hardware–Software Cosynthesis of Distributed Embedded Systems Robert P. Dick, *Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE,* IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 17, NO. 10, OCTOBER 1998