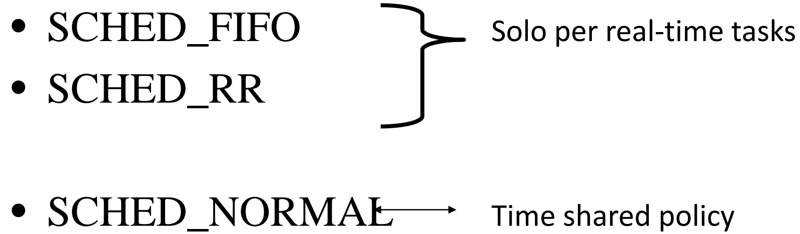
Optimization

Task allocation
Load Balancing
Case study: Networking Applications

Scheduling in General Purpose Multiprocessor Systems

- Runtime task allocation and migration decisions for load balancing
- Well developed for SMP kernels
- Linux O(1) scheduler
  - Overcomes previous work-stealing approach

Linux: Algoritmi di scheduling



Time shared policy

- Il CPU time è diviso in SLICE di tempo, per una esecuzione multiplexing dei processi.
- Le SLICE vengono assegnate in base alla priorità.

Priorità in Linux

Priorità statica.

- Usata per il calcolo del base time quantum (BTQ)
- range da 0 -139 (max-min)
- 0-99 priorità escusiva per kernel tasks
- 100 a 139 priorità user-space tasks

Nice.

- Traduzione della priorità statica per lo user
- Range da -20 - 19 (max min)

Priorità dinamica.

- Usata per lo scheduling della CPU
- Favorisce task interattivi da quelli batch

Base Time Quantum

- Il base time quantum è il tempo massimo per il quale un processo può trattenere la CPU.
- Terminato il BTQ il task va in attesa e dovrà aspettare che tutti i processi presenti in runqueue terminino il loro BTQ.

$$BTQ = \begin{cases} (140 - static_priority) \times 20 & \text{se } static_priority < 120 \\ (140 - static_priority) \times 5 & \text{se } static_priority \geq 120 \end{cases}$$

Alcuni esempi

Priorità	static_priority	valore nice	BTQ	DELTA
Massima priorità	100	-20	800 ms	-3
Alta priorità	110	-10	600 ms	-1
Default	120	0	100 ms	+2
Bassa priorità	130	+10	50 ms	+4
Minima priorità	139	+19	5 ms	+6

Runqueue 2.6.21

- La runqueue è la struttura dati che contiene tutte le informazioni necessarie per l’attività di scheduling della CPU
- In un sistema SMP ad ogni CPU corrisponde una propria runqueue.

Informazioni principali nella runqueue

- Carico del processore
- Puntatore agli array active e expired

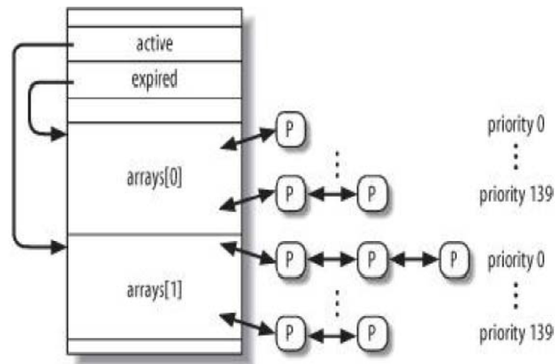
  Le code active e expired sono dei speciali array di priorità che sono alla base del nuovo scheduler 2.6.x
- Puntatore al process descriptor del processo corrente e del processo swapper (PID 0)
- Flag di bilanciamento del carico

Array di priorità

Contengono:

- Il numero totale di tasks nell'array
- Una bitmap delle priorità
- Una coda dubly linked per ogni priorità (140, numerate da 0 a 139)

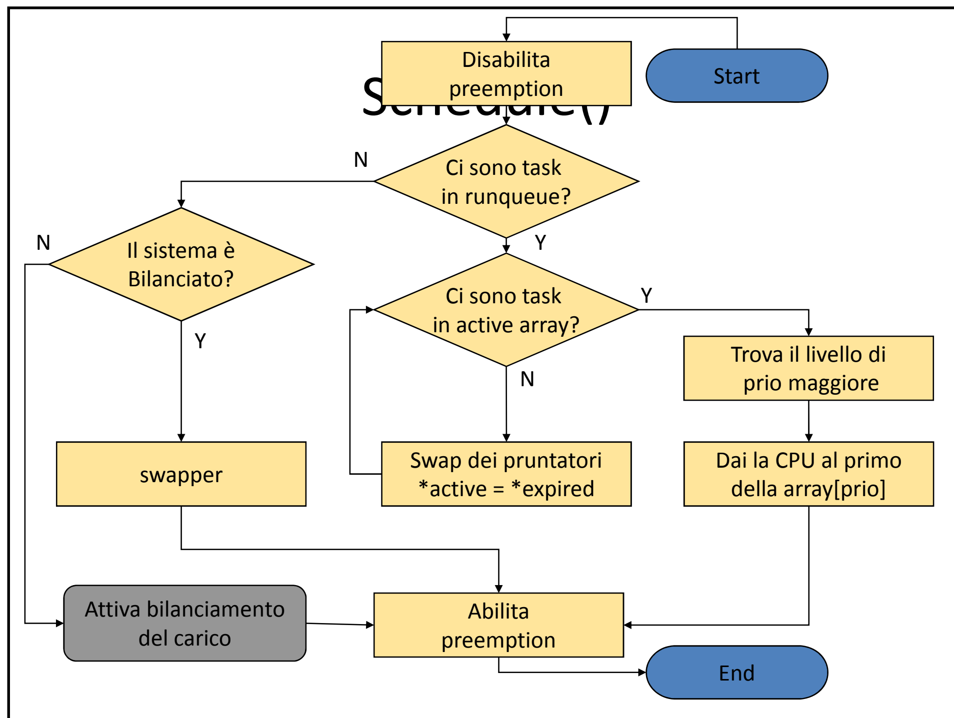
Struttura degli array



Schedule()

Quando viene invocata:

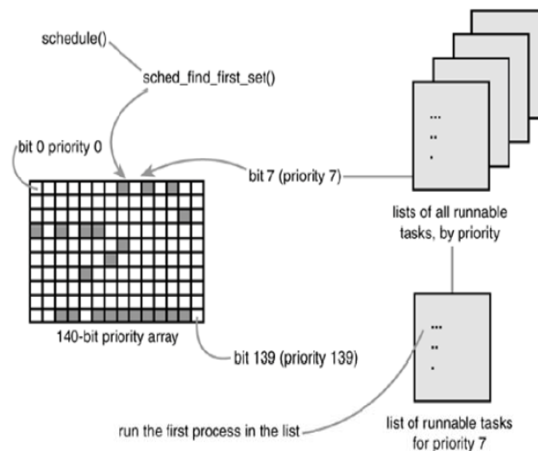
- **Invocazione diretta.** Lo scheduler viene invocato direttamente quando una risorsa di cui a bisogno il processo che si trova in CPU non è più disponibile.
- **Invocazione lazy.**
  - Quando il processo corrente termina il suo BTQ (scheduler_ tick())
  - Quando un processo si sveglia ed ha una priorit`a maggiore del processo corrente (try_to_wake_up())
  - Quando viene invocata la system call sched_setscheduler()



Linux O(1) Scheduler

- Usage of processor local run queues
- Two parameters for scheduling decision
  - Run queue length
  - Cache working set estimates (last execution time)

Algoritmo di scheduling O(1)



Bilanciamento del carico

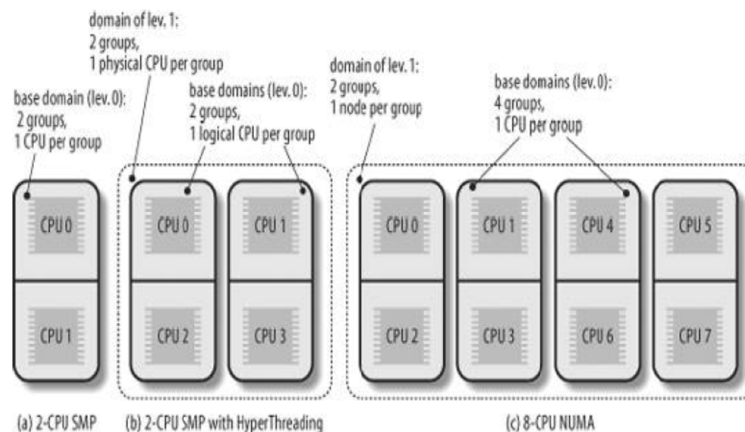
Il bilanciamento del carico, in un sistema SMP viene effettuato dalla funzione in due vie:

- Ogni scheduler_tick(). Ogni tick dello scheduler; viene invocata la funzione run_rebalance_domain(), che si occupa di invocare successivamente load_balance().
- Dalla funzione schedule(), quando una runqueue sta per andare in idle. In questo caso, la funzione schedule invoca una funzione chiamata idle_balance(), la quale successivamente invoca la funzione load_balance_newidle().

Scheduler domains in Linux

- Uno SCHEDULER DOMAINS è un'insieme di CPUs, le quali devo essere mantenute tra loro bilanciate.
  - Ogni scheduler domain divide le proprie CPUs in GRUPPI, dove per sistemi SMP ogni gruppo corrisponde ad una CPU

Esempi scheduler domains



Load_balance()

- Questa funzione viene invocata per ogni runqueue e se trova una ruqueue nel dominio più occupata spinge dei task nella runqueue corrente.

- Come opera:
  1. Invoca find_busiest_group(), ritorna:
     - NULL, è bilanciato;
     - l'indirizzo del gruppo e il numero di tasks da migrare
  2. Invoca find_busiest_queue(), ritorna:
     - l'indirizzo alla runqueue più occupata
  3. Invoca move_task()
     - Se move_task() fallisce viene appeso alla runqueue il flag active_balance = 1, che sveglia il migration kernel thread

Idle_balance()

- E' praticamente identica da load_balance, cambiano solo alcuni parametri per il calcolo del bilanciamento.

- Come opera:
  1. Invoca load_balance_newidle(), una funzione che si comporta come load_balance()
     1. Invoca find_busiest_group(), ritorna:
        - NULL, è bilanciato;
        - l'indirizzo del gruppo e il numero di tasks da migrare
     2. Invoca find_busiest_queue(), ritorna:
        - l'indirizzo alla runqueue più occupata
     3. Invoca move_task()
  2. Se non può spostare nessun task, termina e permette allo scheduler di attivare il processo swapper.

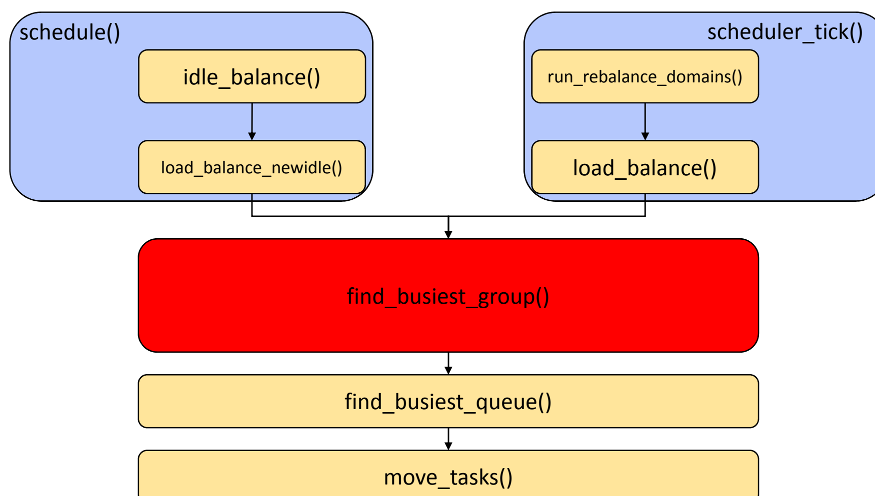
Move_tasks()

- E' la funzione, che in ambo i casi di bilanciamento si occupata di spostare fisicamente dalla runqueue sorgente alla corrente i tasks.

- Argomenti:
  - » this_rq() — puntatore alla runqueue corrente
  - » this_cpu() — cpu corrente
  - » busiest — puntatore alla runqueue più occupata
  - » max_nr_move — numero max di tasks da migrare
  - » ...

- Come opera:
  1. analizza gli array delle priorità, per selezionare i candidati alla migrazione
  2. per ogni candidato invoca can_migrate_task()
     - Parte dall'array dei expired, da quelli con maggiore priorità
  3. se can_migrate_tasks() va a buon fine invoca pull_task()
  4. pull_task():
     1. dequeue_task() — per rimuovere il processo dalla runqueue remota
     2. enqueue_task() — per mettere il processo in coda nella runqueue corrente
     3. resched_task() — per attivare la preemption dei task migrati

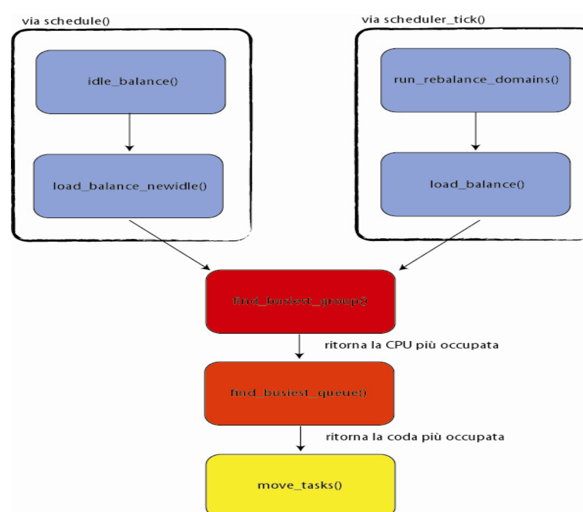
Riepilogo



Find_busiest_group()

- E' la funzione che principalemente gestisce il bilanciamento del carico.
- Essa decide:
  - se il sistema è sbilanciato
  - quale dei gruppi è il più sbilanciato
  - quanti tasks si devo muovere dalla ruqueue sbilanciata alla runqueue corrente

Load Balancing in Linux



Quando una task è migrabile?

- Se il task non è in CPU
- Se la CPU dove si vuole migrare è consentita (`cpus_allowed`)
- Se:
  - il tentativo di migrazione è per molte volte fallito
  - il task non è CACHE HOT

Stima della località (dal Linux Kernel 2.6.8)

"Il task è hot se il tempo trascorso dall'ultima volta in CPU è minore di due volte il costo di migrazione (`cache_hot_time`) di un task generico, con piena località nella CPU corrente alla CPU target, dove esso non ha località"

[Ingo Molnàr, auto-tune migration costs]

```
now - p->last_ran < cache_hot_time
```

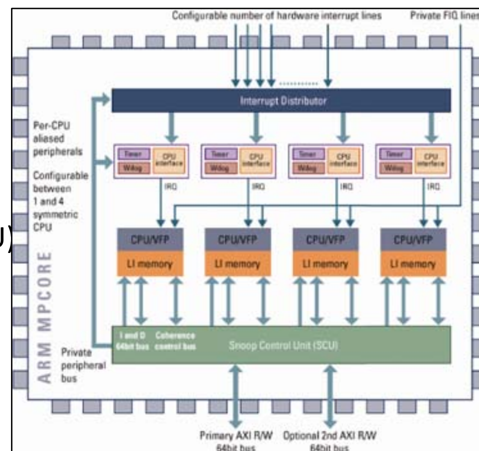
Svantaggi:

- Tiene conto solo del tempo
- Non ha informazioni dirette sulla località

Processore ARM11 MPCore

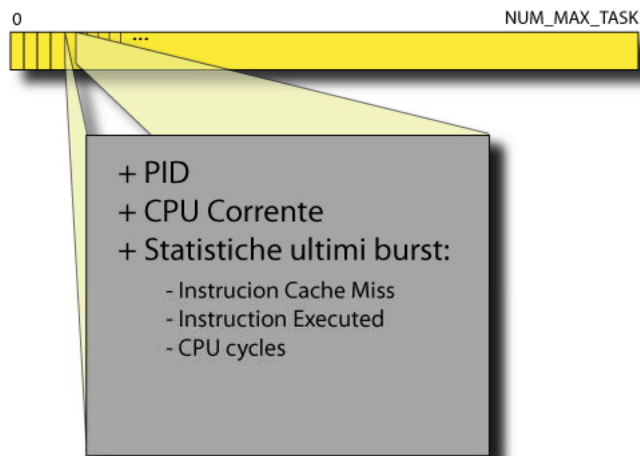
Componenti:

- 4 MP11 core:
  - Integer unit
  - VFP
  - CP15
- Snoop Control Unit (SCU)
- Time & Watchlog
- Distributed Interrupt Controller
- 2 porte AXI 64-bit



Codice evento	Definizione dell'evento
0x00	Cache istruzioni miss su una locazione cachable, e quindi richiesta di fetch da memoria esterna.
0x01	Stallo dovuto al fatto che il buffer non può distribuire una istruzione. Questo può significare una cache istruzioni miss o una Instruction MicroTLB miss.
0x02	Stallo dovuto ad una dipendenza dati.
0x03	Instruction MicroTLB miss.
0x04	Data MicroTLB miss.
0x05	Branch instruction eseguita.
0x06	Branch non predetto.
0x07	Branch mal predetto.
0x08	Istruzione eseguita.
0x09	Folded instruction eseguita.
0x0A	Cache dati accesso in lettura.
0x0B	Cache dati miss.
0x0C	Cache dati accesso in scrittura.
0x0D	Scrittura cache miss.
0x0E	Linea cache sfattata.
0x10	Main TLB miss.
0x11	Richiesta memoria esterna (Cache Refill, Noncachable, Write-Back).
0x12	Stallo dovuto dalla unità di Load Store.
0x13	Evento da LSU.
0x14	Evento da LSU.
0x15	LSU in safe mode.
0xFF	Incremento di un ciclo.

Nuova stima della località



Nuova stima della località

- **`Cache_hot_time` modulato:**

  La soglia cache hot time viene modulata per ogni task rispetto al hit rate dell'utlimo burst in CPU:

  hot_time(t) = cache_hot_time * ultimo hit rate(t)

- **Core Cache Monitor:**
  - Utilizzo delle sole informazioni provenienti dai contatori
  - Un task è caldo se:
    - Cache miss rate dell'ultimo burst < media cache miss rate dei task della CPU
    - Cache miss rate inferiore al 2%
    - E' il solo task in CPU

VM & Linux Scheduler

- When running on a virtual environment, virtual resources are different than physical resources
- Optimal load distribution is computed using information from hypervisor
- Generate low-priority virtual processes with fixed CPU affinity (baloon processes)
- Baloon processes are added to each runqueue

Bounded Multiprocessing (BMP)

- Bounded multiprocessing can be used in association with SMP scheduler such as Linux O(1)
- Some tasks are bounded to processing elements
- They are usually the more intensive tasks
- Other tasks are scheduled by the SMP scheduler

Asymmetric Multiprocessing (AMP)

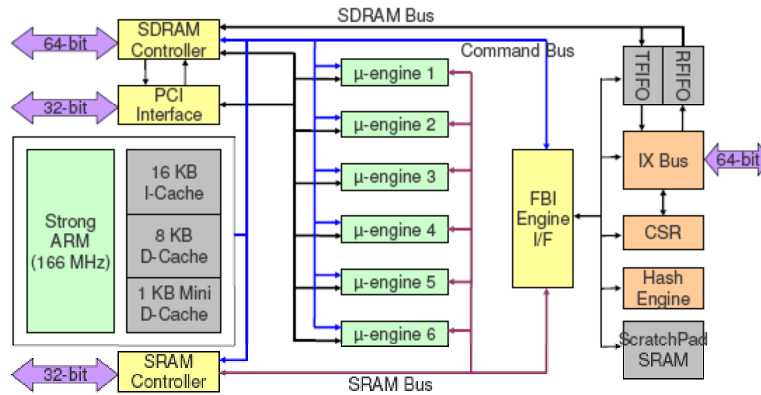
- Two heterogeneous software subsystems over an AMP or SMP architecture
- AMP can be run in a SMP architecture, where one or more cores runs a SMP scheduler (e.g. Linux)
  - Other cores run the more intensive tasks using a microkernel or light runtime support
  - Communication between subsystems through message passing

Scheduling in Multiprocessor Embedded Systems

- Embedded multiprocessor platforms

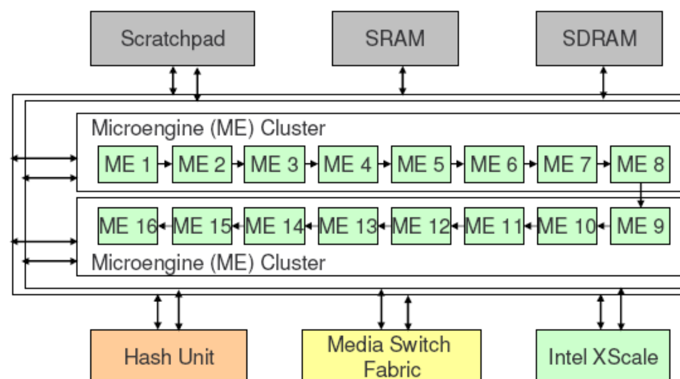
Platform	Company	Area	# Processors
Xenon	Microsoft, IBM	Gaming	3
Cell	IBM, Sony, Toshiba	Gaming	9
IXP2800	Intel	Networking	17
CRS-1 Metro	Cisco	Networking	192
OMAP2420	TI	DSP	4
PC102	PicoChip	DSP	240
Nomadik	STMicro	Mobile	3
MP211	NEC	Mobile	4

Intel IXP1200 Network Processor



- Next generation uses Xscale + 32 u-engines

Intel IXP2800 Network Processor

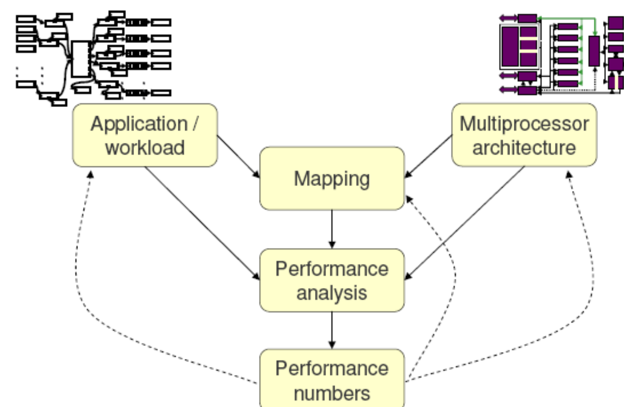


Programming Issues

- Load balanced allocation of packet processing tasks across the different microengines that respects the size of the instruction store
- Assignment of inter-task communications to physical communication links
- Layout of application state across the different memory regions to ensure that tasks have quick access to frequently used data

Deploying concurrent applications

- Y-Chart



Multiprocessor Scheduling Taxonomy

- Partitioning
  - division of tasks in M groups and local scheduling for each processor
  - Simplicity (several uniprocessor scheduling problems)
  - Optimal solution is bin-packing (NP-hard) -> heuristics
  - No migration
  - Global reassignment needed when new task arrive
- Global scheduling
  - Global priority ordered ready queue
  - Task are subject to migration
  - Priority-driven VS dynamic priority (e.g. Pfair schedulers)
  - Pfair schedulers achieve deadline constraints with full utilization
  - ***Ensure global average response time than local queues***

Unbounded Lateness Problem

- Partitioning: impossible to achieve full utilization with bounded lateness
- Example

Resource Allocation Strategies

- Fully static scheduling
  - Require compile time knowledge of job mix
- Fully dynamics scheduling
  - Require actor migration
  - Centralized scheduler
  - Low latency communication between scheduler and processing resources -> limited scalability
- Static assignment
  - Processor allocation done at compile time
  - Local scheduling at runtime

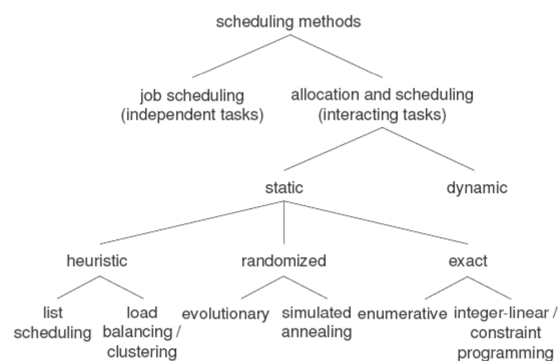
Resource Allocation Strategies

- Resource allocation configurations
  - Optimal scheduling obtained at compile time and stored in a look-up table
  - Number of configurations grows exponentially
  - All jobs must be known at compile time
  - May require actor migration

Global Resource Allocation for Real-Time Streaming Job-Mix

- Resource allocator for HRT streaming on multiprocessors
  - Global admission control + local schedulers
  - Heuristic solution
  - Based on SDF (allow static analysis)
- Approach based on static assignment but with run-time allocation and local scheduling
  - Run-time + steady state phase

Static Scheduling



Problem Modeling: Architecture

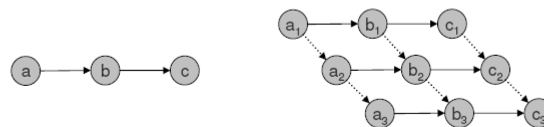
- A common model for the multiprocessor architecture is a network of processors P = {p1, . . . , pm}
- The interconnection network is abstracted as a set of direct point-to-point links between processors
- It is specified by the set C P ×P of the pairs of processors that are connected to each other.
- The pair (p1, p2) 2 C indicates that there is a directed link from processor p1 to processor p2

Problem Modeling: Task Graph Performance

- Common performance model is related to the temporal behavior of the system
- A weight w(v, p) is associated with each vertex or task v 2 V to denote its execution time on a processor p 2 P
- The vertex weight is a function w : V × P ! <+.
- Similarly, weight c((v1, v2), (p1, p2)) along an edge (v1, v2) 2 E denotes the latency or delay due to data transfer when tasks v1, v2 2 V execute on processors p1, p2 2 P, respectively, and (p1, p2) 2 C
- The edge weight is a function c : E × C ! <+, i.e. it is a function of the amount of data transferred along an edge and the nature of physical link to which the edge is assigned

Optimization

- End-to-end delay or makespan
- Throughput:
  - create a new graph G0 that contains multiple iterations of the task graph G and compute a minimum makespan schedule of G0 on the target multiprocessor



Mixed Integer Linear Programming

- Formulation

$\forall v \in V,$

$$x_s(v) \in \Re^+ \quad \text{(start time of task } v\text{)}$$

$\forall v \in V, \forall p \in P,$

$$x_a(v,p) = \begin{cases} 1 & : \text{ if task } v \text{ assigned to processor } p \\ 0 & : \text{ else} \end{cases}$$

$\forall v_1, v_2 \in V, v_1 \neq v_2, (v_1, v_2) \notin E^T, (v_2, v_1) \notin E^T,$

$$x_o(v_1,v_2) = \begin{cases} 1 & : \text{ if task } v_2 \text{ completes after task } v_1 \text{ starts} \\ 0 & : \text{ else} \end{cases}$$

Constraints

- Min max S(v)+w(v)

Allocation constraints

$\forall v \in V,$

$(A1) \quad \sum_{p \in P} x_a(v,p) = 1$

$\forall (v_1, v_2) \in E, \forall (p_1, p_2) \in (P \times P) - C$

$(A2) \quad x_a(v_1, p_1) + x_a(v_2, p_2) \leq 1$

Dependency constraints

$\forall (v_1, v_2) \in E,$

$(S1) \quad x_s(v_2) - x_s(v_1) \geq w(v_1)$

$\forall (v_1, v_2) \in E, \; \forall p \in P,$

$(S2) \quad x_s(v_2) - x_s(v_1) \geq w(v_1) + c((v_1, v_2))(x_a(v_1, p) - x_a(v_2, p))$

Ordering constraints

$\forall v_1, v_2 \in V, (v_1, v_2) \notin E^T, (v_2, v_1) \notin E^T,$

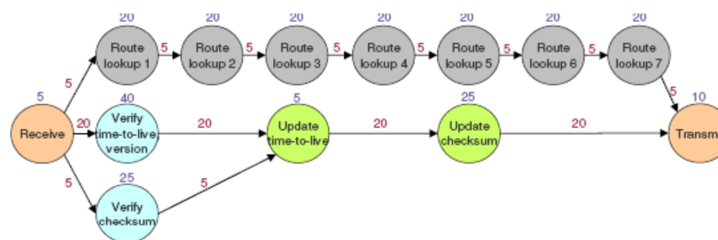
$(O1) \quad x_s(v_2) + w(v_2) - x_s(v_1) \leq M x_o(v_1, v_2)$

$\forall v_1, v_2 \in V, (v_1, v_2) \notin E^T, (v_2, v_1) \notin E^T, \forall p \in P,$

$(O2) \quad x_o(v_1, v_2) + x_o(v_2, v_1) + x_a(v_1, p) + x_a(v_2, p) \leq 3$

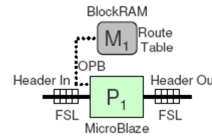
Application Task Graph

- IPv4 routing

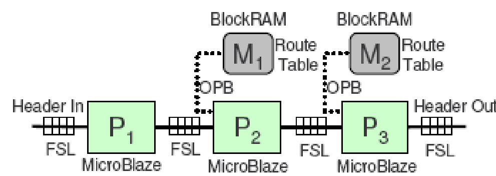


Optimal Allocation using MILP

- Single processor
  - Route table in BRAM
  - 250 cycles/packet
  - 0.2 Gbit



- Single array processor
  - Only P2 and P3 have access to route table
  - P1 controls ingress port, P3 egress port
  - No communication between P1 and P3



Optimum Allocation and Schedule

- M= 165
- Throughput = M/I

  where I is the number of iterations of the task graph
- Upper bound
  - 0.57 Gbits