Lecture 4

Many-core CPU Management and Scheduling

The problem

- Technology allows to design many-core systems by replication. A package consists of 2 cores which share one L2 cache memory.
- The problem is to make most of these resources.

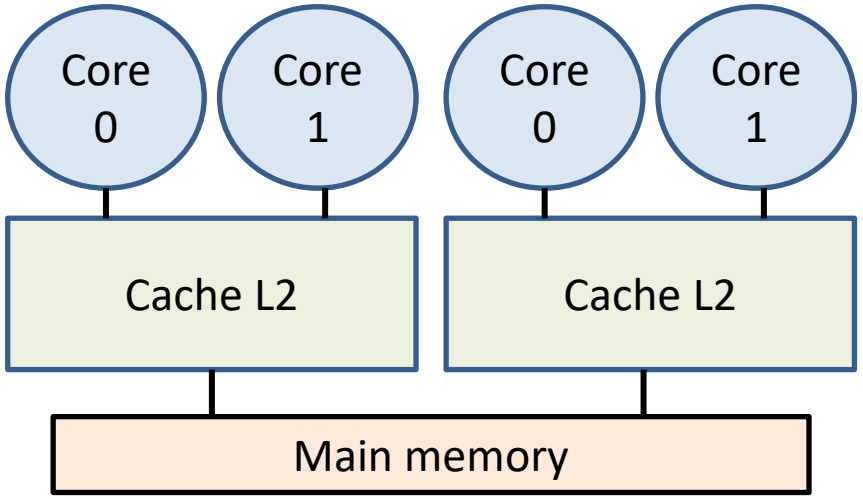


Diagram of 2 packages of 2 cores and L2 each, connected to the main memory

1. Resource mgmt in many-core systems

- With the benefit of parallel execution, the presence of many cores creates new challenges to system designers and developers, including
  - *detecting and programming parallelism;*
  - *cost-effective management of replicated resources;*
  - *dealing with debugging deadlocks and race conditions;*
  - *eliminating performance bottlenecks.*
  - *different levels of granularity require different management mechanisms;*
- *Each application has performance requirements* (QoS) that can be met by a minimal architecture; adding new resources (e.g., cores) can increase the performance above the limit.
- We can define a *virtual architecture* for each application that in terms of resources can be expressed from minimal values to maximal values – performance requirements are translated into architectural requirements (i.e. the virtual architecture).
- Example: VA = {core0, core1, 4 GB mem}

Many-core virtual architecture

- *Spatial component*: allocation of physical resources (cores, L2 cache, pages of memory, buses, etc).

- *Temporal component*: time sharing of resources. It gives the time that resources are dedicated to that virtual architecture / application.

- Resources that are not allocated to any virtual architecture /application are called resources in excess. They can be allocated later for improved performance.

System management

1. Applications have different priorities.
2. The system controls resource allocation among applications - static or dynamic allocation/de-allocation of resources.
3. The system monitors the load of system resources, taking action when a state of overload or underload is detected. The system can provide feedback to applications/users to inform on global resource usage.

- In conclusion, the OS can provide:
  1. VA scheduler;
  2. Spatial partition;
  3. Feedback to the application.

2. Scheduling in many-core systems

- The scheduler main challenge is to identify and predict the resource needs of each process and schedule them in a fashion that will *minimize shared resource contention (e.g., access to L2), maximize shared resource utilization, and exploit the advantage of shared resources between cores.*
- To achieve this, the scheduler needs to be aware of
  1. the topology of the many-core configuration and shared resources,
  2. resource requirements of processes, and
  3. the inter-relationships among processes.
- *Example:* a system with 2 packages, dual-core each package. If the application has two processes (or threads), they can be allocated to different packages, minimising contentions. However, power saving would require to have both allocated to the same package.

Aspects to consider

1. Memory contention (cache L2) and its impact on performance depend on
   - the resources shared;
   - the number of active processes;
   - the *memory access patterns* of the individual processes.

2. Heterogeneous data access patterns of *memory-intensive processes* running on the cores sharing caches can lead to cache contention and sub-optimal performance.

- Approaches:
  1. If processes share data, it makes sense to schedule them on the same package cores. Otherwise, scheduling on one package will lead to L2 contention; the main benefit is power saving from idle packages that can switch both cores and L2 cache to sleep states.

  2. Scheduling processes on cores of different packages maximises execution speed but it’s not optimal in respect to energy.

3. Process group scheduling

- If all processes are resource intensive, the challenge before the scheduler is to identify the processes that *share data* and schedule them on the same package. This will also result in efficient data communication among processes that share data.

- The system software has some inherent knowledge about data sharing among processes:
  - threads belonging to a process share the same address space and as such share everything (text, data, heap, etc.);
  - similarly, processes attached to the same memory segment will share data in that segment.

- In a scenario where all the shared resources and packages are busy, the scheduler needs to minimize the resource contention. For example, *grouping CPU-intensive and memory-intensive processes* onto the cores sharing the same L2 cache will result in minimized cache contention.

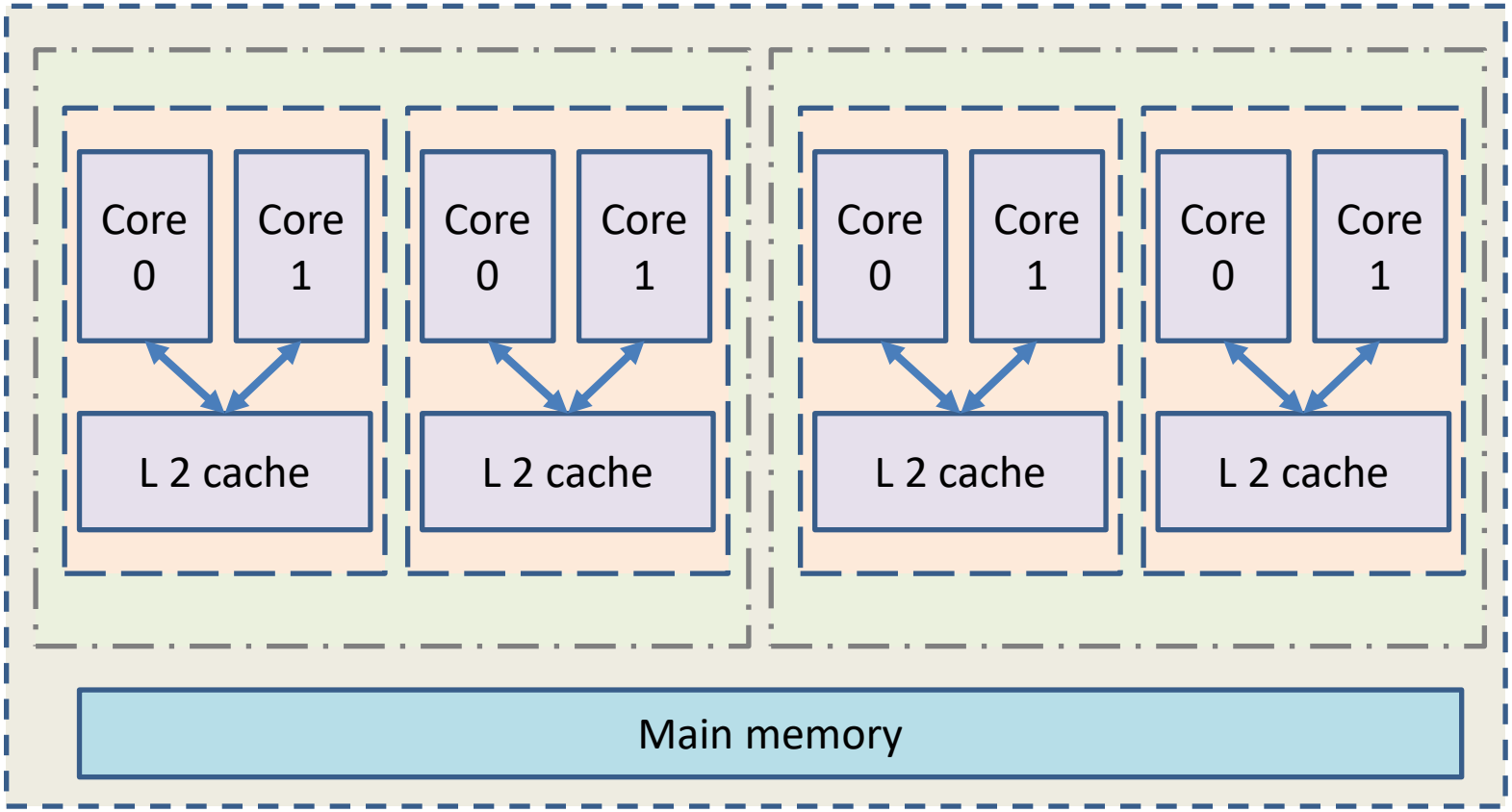
Prediction of resource use

- Process characteristics and behaviour can be predicted using the *micro-architectural history* of a process with performance counters. In the absence of such micro-architectural information, the system software can also use some heuristics to estimate the resource requirements.
- For example, one can use the number of physical pages that are accessed (using the Accessed flag in the page tables that manage virtual to physical address translation in x86 architecture) for certain intervals or use the processes memory Resident Set Size (RSS).
- The process scheduler can use this information and group processes on the cores residing in a physical package with the goal of minimizing shared resource contention.

4. Scheduling domains

- *Load balancing*: in many-core systems, one goal of the scheduler is to balance the cores' load such that there is no idle core while other cores are overloaded.

- The *domain-based scheduler* aims to solve this problem by using of a new data structure which describes the system's architecture and scheduling policy in sufficient detail that good decisions can be made:
  - a *scheduling domain* (struct sched_domain) is a set of cores which share properties and scheduling policies, and which can be balanced against each other. *Scheduling domains are hierarchical*; a multi-level system will have multiple levels of domains.
  - each domain contains one or more *core groups* (struct sched_group) which are treated as a single unit. When the scheduler tries to balance the load within a domain, it tries to even out the load carried by each core group without worrying directly about what is happening within the group.



Hierarchically, there are three domain types:

1. A, a package of two cores and their share L2 cache;
2. B, two packages, four cores;
3. C, 4 packages, eight cores.

Domain parameters

- Each scheduling domain has parameters which control how decisions are made at that level of the hierarchy. The main parameters are:

1. *Balance frequency*: how often attempts should be made to balance loads across the domain;

2. *Unbalance*: how far the loads on the component cores are allowed to get out of sync before a balancing attempt is made;

3. *Idle process*: how long a process can sit idle before it is considered to no longer have any significant cache affinity;

4. various policy flags.

Policy examples

1. When a process calls exec() to run a new program, its current cache affinity is lost. At that point, it may make sense to move it elsewhere. So the scheduler works its way up the domain hierarchy looking for the highest domain which has the SD_BALANCE_EXEC flag set. The process will then be shifted over to the core within that domain with the lowest load. Similar decisions are made when a process forks.
2. If a core becomes idle, and its domain has the SD_BALANCE_NEWIDLE flag set, the scheduler will go looking for processes to move over from a busy core within the domain.
3. If one core in a shared pair is running a high-priority process, and a low-priority process is trying to run on the other core, the scheduler will actually idle the second core for a while. In this way, the high-priority process is given better access to the shared cache.

Policies pseudo-code

1. if exec() is invoked then
   - determine the highest domain with load balancing flag set;
   - move the process to the least loaded core of the domain;

2. if core is idle then
   - determine overloaded core;
   - move load to the idle core;

3. if priorities of processes running on the two core are high/low then
   - stop the low priority process;

5. Active balancing

- The last component of the domain scheduler is *the active balancing code*, which moves processes within domains when things get too far out of balance.

- Every scheduling domain has an interval which describes how often balancing efforts should be made; if the system tends to stay in balance, that interval will be allowed to grow. The scheduler "rebalance tick" function runs out of the clock interrupt handler; it works its way up the domain hierarchy and checks each one to see if the time has come to balance things out. If so, it looks at the load within each core group in the domain; if the loads differ by too much (unbalance), the scheduler will try to move processes from the busiest group in the domain to the least busy group. In doing so, it will take into account factors like the cache affinity time for the domain.

Questions

1. What can we consider as the load of a core, the number of processes ready to run, or %CPU usage? How frequently should the core load be assessed?
2. In your opinion, is the priority of processes playing any role in the allocation of resources to virtual architectures?
3. In your opinion, is time sharing of resources allocated to virtual architectures a good decision all the time?
4. Is resource replication having a good or a bad impact on energy saving?

Problem

1. Consider a homogenous 8-core computer. The basic domain unit is a 2-core package with L2. Draw the binary-tree model of the domain hierarchy up to the root level that includes all cores. Define policies for each domain, considering energy saving and load balancing as the main criteria. Write the domain scheduler that works with this model using pseudo-code.

References

- http://www.intel.com/technology/itj/2007/v11i4/9-process/2-intro.htm
- http://lwn.net/Articles/80911/
- https://www.androidauthority.com/fact-or-fiction-android-apps-only-use-one-cpu-core-610352/