Cache Partitioning

February 15, 2013

1 Problem definition

In the SMP system, multiple processes can run simultaneously, interfering each other on shared resources like LLC. Our goal is to provide performance isolation among co-existing processes and prove that cache can be used in real-time systems for boosting performance without compromising real-time properties.

The first base line: multiple benchmark applications (more than the number of cores) run on Linux, sharing LLC, no isolation. We can repeat this experiment hundreds of times to get the performance variation and the observed worst case execution time for each benchmark application.

Another base line: multiple applications run on Linux, with LLC disabled. We get the average performance for each application.

Our approach: the same benchmark applications run on modified Linux, with page coloring and cache-aware scheduler. Again, repeat the experiment to get the average performance, performance variation and observed worst case execution time. We can also show the system wide LLC misses reduced.

2 Approaches

2.1 page coloring

We profile each benchmark application to get the appropriate number of colors offline, which should not be too large even if the optimum is the whole LLC. Colors from different processes can overlap (we may also try non-overlapping). We can begin with simple round robin approach, according to the result, a smart approach to overlap colors may be needed. Then we will run applications with static color assignment.

2.2 Cache-aware scheduling

we can test different degree of LLC isolation in scheduler.

The first one is complete isolation. So each time when the scheduler tries to pick a new task, it can have two policies:

1. Pick a task as usual, then check whether it has color conflict with running tasks; if has, put it back to runqueue and schedule idle task to run until another scheduling event happens on another core.
2. Pick a task as usual, then check whether it has color conflict with running tasks; if has, pick the next task in the runqueue and check again; repeat this until a task without color conflict is found (we can limit the number of repetitions to reduce scheduling overhead and the impact on fairness). If no task is found, pick idle task.

The second one allows scheduler to introduce limited cache interference. So when the scheduler picks a task as usual, as long as it doesnt have too much color conflict (for example, more than 2 overlapping colors), it will be scheduled up. If it has, use the policies in the first scheme above.

3 Implementation

We will have a loadable kernel module. If it's not loaded, everything works as original Linux. After it gets loaded, newly created processes will be assigned address space to different portions of LLC.

3.1 Color assignment

In order to keep track of color assignment, a data structure is inserted into memory descriptor.

Before the kernel module is loaded, every process has zero color assignment when it is created.

Real work happens after the kernel module gets loaded. When a process is forked, since it's still sharing resource with its parent, it inherits color assigment from parent. This inheriting is done in fork(). Later, if exec() is called when starting a new program, color assignment function from the kernel module will be invoked to assign page colors to the calling process. Policy for color assignment is implemented in the kernel module.

3.2 Page allocator

When the kernel module is loaded, it creates a memory pool of predefined size for each page color. This is simply managed by singly-linked lists. The memory pool has to be large enough to serve all memory requests from benchmark applications running in the experiment.

Some checking code is inserted into page fault handler. When a user space page fault occurs, it checks whether the process causing this fault has a non-empty color set. If it doesn't have, the original Linux page allocation method is called; if it has, then page allocation function from the kernel module is called, which allocates a page according to the assigned color set in a round robin fashion.

Currently, only pages from anonymous memory regions (stack and heap) or Copy-On-Write pages are colored.

3.3 Scheduler

Cache-aware scheduler is achieved by modifying the CFS scheduler. Every time when schedule() is called, a piece of code is added to check the process picked by scheduler to run as the next task. The color set of this process is checked against running processes on other cores to see whether it has color conflict with them. The way to handle color conflict in scheduling has been described before.

Implementation is still under development.

4 Plan

I would like to profile benchmark applications within a few weeks and try to identify those that will have heavy cache interference with each other when running together. Next, I am going to modify the scheduler to be cache aware and check the application performance improvement. Based on the result, I may try some other policies for color assignment or scheduling.