Co-scheduling HPC workloads on cache-partitioned CMP platforms

Anne Benoit
ENS Lyon, France & Georgia Tech, USA
Anne.Benoit@ens-lyon.fr

Joint work with Guillaume Aupy, Yves Robert, Brice Goglin and Loic Pottier

CCDSC 2018 - La Maison des Contes, France
September 4-7, 2018

My 10th year anniversary as well, thanks Jack & Bernard ☺

Why co-scheduling?

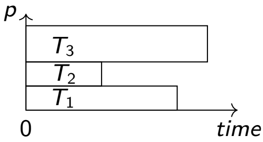
- Chip multiprocessors (CMP): increasing number of cores
- Most of parallel applications are not perfectly parallel (communication overhead, etc)
- Large-scale simulations: *in-situ* and *in-transit* processing problematics, i.e., simulations and data analysis are running concurrently

Why co-scheduling?

- Chip multiprocessors (CMP): increasing number of cores
- Most of parallel applications are not perfectly parallel (communication overhead, etc)
- Large-scale simulations: *in-situ* and *in-transit* processing problematics, i.e., simulations and data analysis are running concurrently

Solution: increase platform efficiency by **concurrently scheduling** parallel applications! ☺

Co-Scheduling [Ousterhout, 1982]: Execute multiple tasks **at the same time** on the same platform, in order to maximize platform throughput

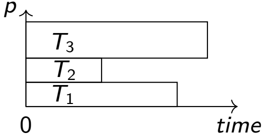


Why co-scheduling?

- Chip multiprocessors (CMP): increasing number of cores
- Most of parallel applications are not perfectly parallel (communication overhead, etc)
- Large-scale simulations: *in-situ* and *in-transit* processing problematics, i.e., simulations and data analysis are running concurrently

Solution: increase platform efficiency by **concurrently scheduling** parallel applications! ☺

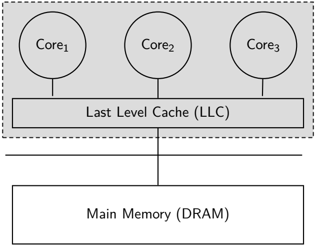
Co-Scheduling [Ousterhout, 1982]: Execute multiple tasks **at the same time** on the same platform, in order to maximize platform throughput



But the remedy comes with complications: **co-run degradation** ☹

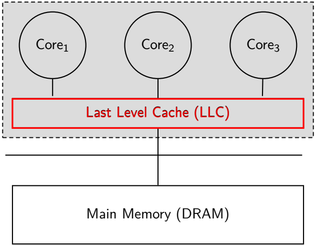
Why partitioning the cache?

- In CMP caches, prefetching units are shared between cores
- Multiple applications (i.e., co-schedule) running on a CMP may create interferences on shared resources
- This work only focuses on interferences at the last-level cache



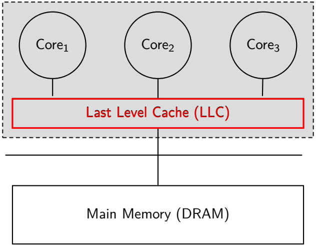
Why partitioning the cache?

- In CMP caches, prefetching units are shared between cores
- Multiple applications (i.e., co-schedule) running on a CMP may create interferences on shared resources
- This work only focuses on interferences at the last-level cache



Why partitioning the cache?

- In CMP caches, prefetching units are shared between cores
- Multiple applications (i.e., co-schedule) running on a CMP may create interferences on shared resources
- This work only focuses on interferences at the last-level cache



The proposed solution is to use a **cache-partitioning** approach

Optimization problem

Execute m iterative applications $A_1, \ldots, A_m$ on P identical cores

These P cores are sharing a cache of size C

This cache of size C can be divided into X slices (cache fractions)

Optimization problem

Execute m iterative applications $A_1, \ldots, A_m$ on P identical cores

These P cores are sharing a cache of size C

This cache of size C can be divided into X slices (cache fractions)

How many cores and how many cache fractions should we give to each application for an efficient execution and use of the platform?

Optimization problem

Execute m iterative applications $A_1, \ldots, A_m$ on P identical cores

These P cores are sharing a cache of size C

This cache of size C can be divided into X slices (cache fractions)

How many cores and how many cache fractions should we give to each application for an efficient execution and use of the platform?

- What we can play on:
  - p_i: number of cores on which application A_i is executed
  - x_i: number of cache fractions assigned to A_i
- Constraints:
  - $\sum_{i=1}^{m} p_i = P$
  - $\sum_{i=1}^{m} x_i = X$

Execution time

- Amdahl's law [1967]: $t_i(p_i) = s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}$, where
  - s_i is the sequential fraction of A_i ($0 =$ perfectly parallel)
  - T_i^{seq} is the sequential computation time of A_i
  - p_i is the number of cores used by A_i

Execution time

- Amdahl's law [1967]: $t_i(p_i) = s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}$, where
  - s_i is the sequential fraction of A_i (0 = perfectly parallel)
  - T_i^{seq} is the sequential computation time of A_i
  - p_i is the number of cores used by A_i

- Time further impacted by cache misses: slowdown based on Power Law of cache misses [Harstein'08]; cache miss ratio r of a cache of size C_{act} expressed as $r = r_0 \left(\frac{C_0}{C_{act}}\right)^{\alpha}$, where
  - r_0 is the cache miss ratio for a baseline cache of size C_0
  - α is a parameter ranging from 0.3 to 0.7 ($\alpha = 0.5$)

Execution time

- Amdahl's law [1967]: $t_i(p_i) = s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}$, where
  - s_i is the sequential fraction of A_i ($0 =$ perfectly parallel)
  - T_i^{seq} is the sequential computation time of A_i
  - p_i is the number of cores used by A_i
- Time further impacted by cache misses: slowdown based on Power Law of cache misses [Harstein'08]; cache miss ratio r of a cache of size C_{act} expressed as $r = r_0 \left(\frac{C_0}{C_{act}}\right)^{\alpha}$, where
  - r_0 is the cache miss ratio for a baseline cache of size C_0
  - α is a parameter ranging from 0.3 to 0.7 ($\alpha = 0.5$)
- Overall, $T_i(p_i, x_i) = \underbrace{\left(s_i T_i^{seq} + (1 - s_i)\frac{T_i^{seq}}{p_i}\right)}_{\text{Computations}} \times \underbrace{\left(c_i + \frac{b_i}{\sqrt{x_i}}\right)}_{\text{Slowdown}}$
  - x_i is the fraction of cache given to A_i
  - b_i and c_i are some constants pertaining to A_i

Optimization problem

- Time to compute one iteration of A_i, given p_i and x_i
- CoSched-CachePart: maximize the **weighted** throughput
- *in-situ* and *in-transit* mentioned in introduction: we do not want data analysis phase slowing down the simulation

Optimization problem

- Time to compute one iteration of A_i, given p_i and x_i
- CoSched-CachePart: maximize the **weighted** throughput
- *in-situ* and *in-transit* mentioned in introduction: we do not want data analysis phase slowing down the simulation

- β_i denotes the weight of application A_i, i.e., the number of times that we should execute A_i at each iteration step
- For instance, A_1 and A_2 with $\beta_1 = \frac{1}{4}$ and $\beta_2 = 1$:
  - we execute A_1 only once every four steps
  - we execute A_2 at each step, hence four executions of A_2 for one execution of A_1
- Note that $(\beta_1 = \frac{1}{4}, \beta_2 = 1)$ is equivalent to $(\beta_1 = 1, \beta_2 = 4)$

Optimization problem

- Time to compute one iteration of A_i, given p_i and x_i
- CoSched-CachePart: maximize the **weighted** throughput
- *in-situ* and *in-transit* mentioned in introduction: we do not want data analysis phase slowing down the simulation

- β_i denotes the weight of application A_i, i.e., the number of times that we should execute A_i at each iteration step
- For instance, A_1 and A_2 with $\beta_1 = \frac{1}{4}$ and $\beta_2 = 1$:
  - we execute A_1 only once every four steps
  - we execute A_2 at each step, hence four executions of A_2 for one execution of A_1
- Note that $(\beta_1 = \frac{1}{4}, \beta_2 = 1)$ is equivalent to $(\beta_1 = 1, \beta_2 = 4)$

$$\text{Maximize} \min_{1 \le i \le m} \left\{ \frac{1}{\beta_i T_i(p_i, x_i)} \right\} \quad \text{subject to} \left\{ \begin{array}{l} \sum_{i=1}^m p_i = P \\ \sum_{i=1}^m x_i = X \end{array} \right.$$

Scheduling strategies: DP-CP

We can solve the CoSched-CachePart problem **optimally**, with a dynamic programming algorithm ☺

Theorem 1

CoSched-CachePart can be solved in time $O(mPX)$, where m is the number of applications, P is the number of processors, and X is the number of different possible cache fractions.

$T(i, q, c)$ is the maximum weighted throughput with $A_1, \dots, A_i$, using q cores and c fractions of cache:

$$T(i,q,c) = \begin{cases} \displaystyle\max_{\substack{1 \le q_1 \le q \\ 1 \le c_1 \le c}} \frac{1}{\beta_1 T_1(q_1, c_1)} & \text{if } i = 1, \\ \displaystyle\max_{\substack{1 \le q_i < q \\ 1 \le c_i < c}} \left\{ \min \left\{ T(i-1, q-q_i, c-c_i), \frac{1}{\beta_i T_i(q_i, c_i)} \right\} \right\} & \text{otherwise.} \end{cases}$$

Scheduling strategies

- DP-CP: optimal dynamic programming algorithm
- Eq-CP: same number of cores and the same number of cache fractions to each application:
  - First give $p_i = \left\lfloor \frac{P}{m} \right\rfloor$ and $x_i = \left\lfloor \frac{X}{m} \right\rfloor$ to each A_i
  - Next give $P \mod m$ extra cores one by one to the **first** $P \mod m$ applications; $X \mod m$ extra cache fractions one by one to the **last** $X \mod m$ applications
- DP-Equal: same number of cores as DP-CP, but shares cache equally across applications as Eq-CP

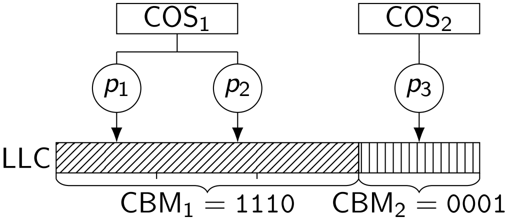
Scheduling strategies

- DP-CP: optimal dynamic programming algorithm
- Eq-CP: same number of cores and the same number of cache fractions to each application:
  - First give $p_i = \left\lfloor \frac{P}{m} \right\rfloor$ and $x_i = \left\lfloor \frac{X}{m} \right\rfloor$ to each A_i
  - Next give $P \mod m$ extra cores one by one to the **first** $P \mod m$ applications; $X \mod m$ extra cache fractions one by one to the **last** $X \mod m$ applications
- DP-Equal: same number of cores as DP-CP, but shares cache equally across applications as Eq-CP

- Two variants where cache partitioning is disabled (all applications access 100% of the LLC):
  - DP-NoCP uses the same number of cores as DP-CP
  - Eq-NoCP uses an equal-resource assignment as in Eq-CP

Experimental setup

- Two Intel Xeon E5-2650L v4 Broadwell, 14 cores each, Hyper-Threading disabled
- 35MB last-level cache divided into 20 slices
- Vanilla 4.11.0 Linux kernel with cache partitioning enabled
  - Cache Allocation Technology (CAT): Provided by Intel to partition the last-level cache (LLC)
  - Part of the Resource Director Technology (RDT)
  - Class of services (COS), with 4-bit capacity mask (CBM)
- Example where first COS has 2 cores and 75% of LLC:



Model accuracy

- Instantiate the model and check its accuracy
- Three applications from NAS Parallel benchmarks with shared memory (class=A): CG (conjugate gradients), MG (multi-grid solve), FT (discrete 3D fast Fourier Transform)
- a_i, b_i, and s_i are obtained by interpolation from the data produced by measurements

App_i	$a_i(= c_i - 1)$	b_i	s_i
CG	-0.0379	0.0474	0
MG	0.0460	0.0073	0.065
FT	0.0092	0.0129	0.016

Relative error defined as

$$E_i(p_i, x_i) = \frac{\left| T_i(p_i, x_i) - T_i^{real}(p_i, x_i) \right|}{T_i^{real}(p_i, x_i)},$$

where $T_i^{real}(p_i, x_i)$ is the measured execution time for A_i

How to instantiate the model?

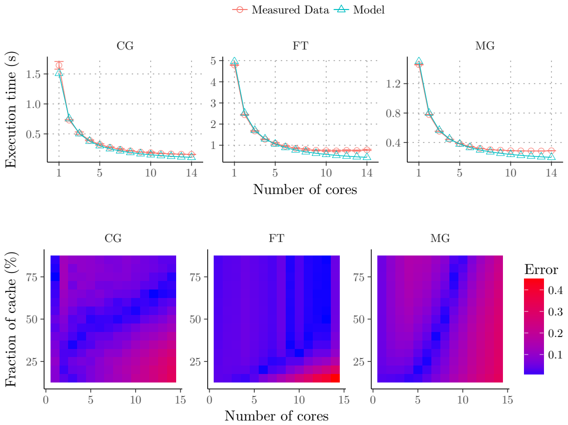
We need to find the three constants per application: s_i, a_i and b_i

- We monitor each application with PAPI to record cache miss ratio, execution time, etc
- Each application A_i executes alone on a dedicated processor to avoid perturbations

- For s_i: we give 100% of the cache to the application A_i and vary the number of cores from 1 to 14
- For a_i and b_i: we record cache misses for $15\% \leq x_i \leq 85\%$ and $1 \leq p_i \leq 14$ (280 values used)

From these data, we use interpolation method to obtain the best s_i, a_i and b_i for each A_i

Model accuracy (cache fraction = 15% for the 1^{st} figure)



- The model is accurate, in particular with enough cache fractions and not too many processors
- Simplifying assumptions not completely true in practice (cache misses independent of number of cores)

Experimental results

- We modified the main loop of NAS applications such that each of them computes for a duration T
- We ensure that each application reaches the steady state with enough iterations ($T = 3$ minutes)
- We use 12 cores instead of 14 to avoid rounding effects
- The platform has two processors: one is used to run the experiments, the other manages the experiments (cache experiments are highly sensitive)

In this talk, we focus on CG and MG, since it is the most interesting combination in terms of cache partitioning

Results: Metrics

We measure the time for one iteration of A_i:

$$T_i = \frac{T}{\#\texttt{iter}_i}$$

where $\#\texttt{iter}_i$ is the number of iterations of application A_i during T.

Weighted throughput

We want to maximize:

$$\min_i \frac{1}{\beta_i T_i}$$

Distance to the optimal fairness (goal: all $\beta_i T_i$'s equal)

$$\Delta_{fairness} = \sum_{i \neq j} \left| \frac{\beta_i T_i}{\beta_j T_j} - 1 \right|$$

Results: Impact of cache partitioning

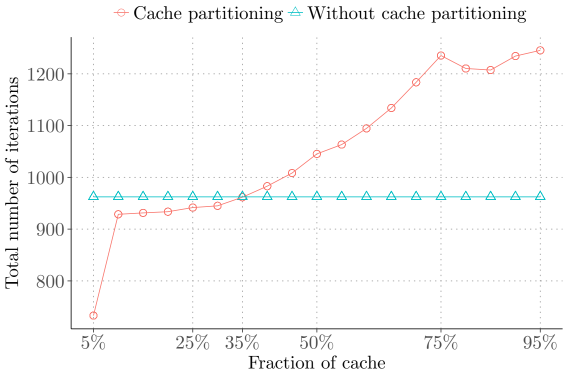


Figure: CG and MG (six cores each). Cache fraction of *CG* varying from 5% to 95%.

Cache partitioning can help! In particular when compute-intensive and communication-intensive applications are co-scheduled

Results with two applications

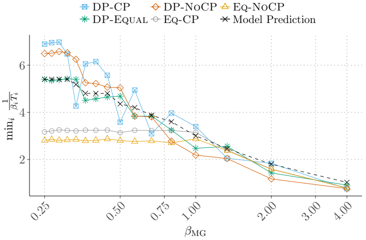
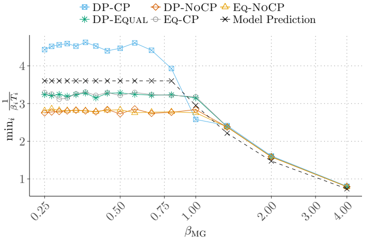


Figure: Minimum throughput and $\Delta_{fairness}$ for CG and MG.

- DP-CP outperforms DP-NoCP, cache partitioning provides a good performance improvement
- DP-* have a $\Delta_{fairness}$ close to zero, while EQ-* are further from optimal fairness
- Model accurate enough (analytical throughput from $T_i(p_i, x_i)$)

Results with two applications (each with 6 cores)



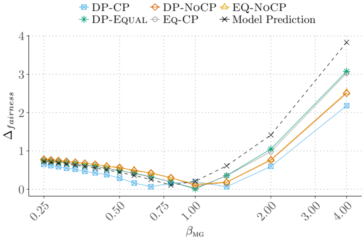


Figure: Minimum throughput and $\Delta_{fairness}$ for CG and MG, where both applications have six cores.

- We can clearly see the impact of cache on performance here (DP-CP is the best).
- Up to 25% improvement when $\beta_{\mathrm{MG}} < 1$

Results with three applications

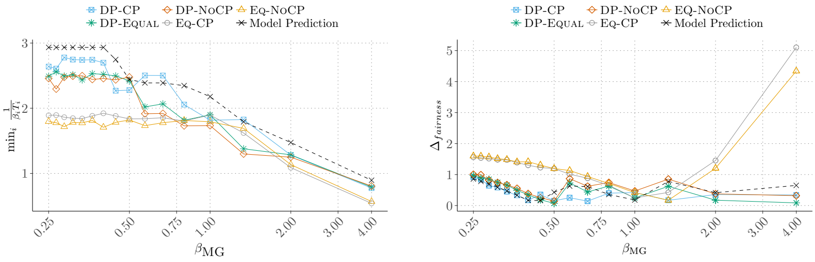


Figure: Minimum throughput and $\Delta_{fairness}$ for 2CG+MG.

- DP-CP exhibits a **gain** around 15% on average over DP-NoCP and DP-Equal!
- Model even more accurate than with two applications

Conclusion

- Model for the execution time of each application
- Instantiate the model on applications coming from the NAS benchmarks
- The model is accurate: comparison between predicted execution time and measured execution time

- Several scheduling strategies have been designed
- Real experiments using CAT
- In practice, optimal strategy often leads to better results than equal sharing of resources or no cache partitioning

- Which combinations of applications benefit most from cache partitioning? Co-schedule of compute-intensive applications (CG) with memory-intensive one (MG)

Future work

- Confirm the usefulness of cache partitioning on a larger platform
- Design a better interpolation strategy, capable of retro-fitting a subset of the experimental data
- Generalize the experiments to multiprocessors (moving applications from one processor to another)