Empirical Study of Cache-Adaptivity of Cache-Oblivious Algorithms

Project Report

By Group 17
Sagar Shah (schshah)
Raunaq Kochar (rakochar)

Problem description (+ Motivation)

For over two decades, practitioners have recognized the desirability of algorithms that adapt as the availability of RAM changes. Such changes naturally occur as processes come and go on the same system and compete with each other for memory resources. Database researchers, in particular, have published empirically efficient adaptive sorting and join algorithms.

When multiple programs are running in parallel, since the hardware cache space is shared among the processes, the availability of the cache is reduced for each program and each program becomes slower. We want to come up with good work loads and simulate real time scenarios to empirically determine how the running time of a program changes, when its cache space is shared with multiple programs.

Cache adaptive programs adapt to changes in cache space i.e. as the cache space reduces, the program becomes slower, but if the program isn't cache-adaptive, it will become very slow.

Programmers cannot allocate space on the cache. Unless a program is running alone, there is no possible way to know how much cache space is available to each program. If two threads of a process are doing different tasks, they will themselves have conflicts over cache space and if they both require a lot of cache space, they will constantly incur cache misses which will result in loss of bandwidth.

In [1], it has been theoretically proved that some cache oblivious algorithms are cache adaptive and some are not. No one has done extensive experiments to show how cache adaptivity affects running time. In [2], it has been shown that cache adaptivity can affect running time a lot using some empirical experiments on algorithms such as:

- Parenthesis Problem.
- Gap Problem
- Floyd-Warshall's APSP.

These experiments were done by only changing the cache size but without the change in available cache-I/O data transfer bandwidth.

Goals

Proposed Goals

- In this project, implement a tool that:
    - Change available cache space on the fly based on some given schedule that the user can specify.
    - When the cache space changes, effective bandwidth for cache-I/O data transfer changes accordingly for the algorithm under analysis.
- Perform experiments on many algorithms based on real time schedules in changing cache availability and effective memory bandwidth and collect various parameters like time of execution, cache miss ratio, fetch ratio, energy consumption.

Goals Achieved

- Case study on various tools available that could:
    - Change available cache space
    - Change effective available bandwidth between cache and memory
    - Measure cache hits/misses, miss ratio/fetch ratio
    - Case study on: Cache Pirate, PAPI, Linux Cgroups, Cache Grind, Likwid, StressMark
- Developed an algorithm and tool that:
    - Is used to change cache and I/O bandwidth available to the target program as per requirement mentioned above
    - Provides user configurable execution environment as close to real environment for performance measurement.
- Execution time after running various algorithms was collected.
- It is possible to collect the fetch ratio, miss ratio etc data by CBpirate, however, our system did not support collection of such data.

Prior Work

Cache Oblivious Algorithms:

No variables dependent on hardware parameters, such as cache size and cache-line length, need to be tuned to achieve optimality.

There exist a general class of *cache-oblivious* algorithms that are optimally *cache-adaptive.*

Cache Adaptive Model:

The cache-adaptive (CA) model is an extension of the DAM and ideal cache models. The CA model describes systems in which the available memory can change dynamically. In the CA model, the algorithm has the following parameters:

- ***M(t)*****:** size of the cache.

- ***B:*** size of cache line.(unit of data transfer between cache and memory.)

The performance of a CA model is measured in terms of I/Os.

In CA model A is considered a competitive algorithm to A', if A has c-*speed augmentation* compared to A' then A performs its $(c*t)^{th}$ I/O when A' performs its t^{th} I/O. An algorithm A that solves problem P is optimal in the cache-adaptive model if there exists a constant c such that on all memory profiles and all sufficiently large input size N, the worst-case running time of a c-speed augmented A is better than the worst-case running time of any other (non-augmented) algorithm.

In [1], it has been theoretically proved that some cache oblivious algorithms are cache adaptive and some are not. No one has done extensive experiments to show how cache adaptivity affects running time. In [2], it has been shown that cache adaptivity can affect running time a lot using some empirical experiments on algorithms such as:

- Parenthesis Problem.
- Gap Problem
- Floyd-Warshall's APSP.

These experiments were done by only changing the cache size but without the change in available cache-I/O data transfer bandwidth.

Case study

1. **Likwid**
    - likwid-perfctr: configure and read out hardware performance counters on Intel and AMD processors
    - likwid-pin: pin your threaded application (pthread, Intel and gcc OpenMP to dedicated processors)

2. **Cache Grind**
   Cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor. It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines. However, some modern machines have three or four levels of cache. For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and last-level caches.
    - Disadvantages:
        - Simulation may not alway be similar to real hardware
        - It's slow

3. **Linux CGroups:**
    - Stands for Linux Control Groups.
    - Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

- Cgroups allow you to allocate resources — such as CPU time, system memory, network bandwidth, or combinations of these resources — among user-defined groups of tasks (processes) running on a system.
- You can monitor the cgroups you configure, deny cgroups access to certain resources, and even reconfigure your cgroups dynamically on a running system.
- By using cgroups, system administrators gain fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. Hardware resources can be appropriately divided up among tasks and users, increasing overall efficiency.

4. PAPI

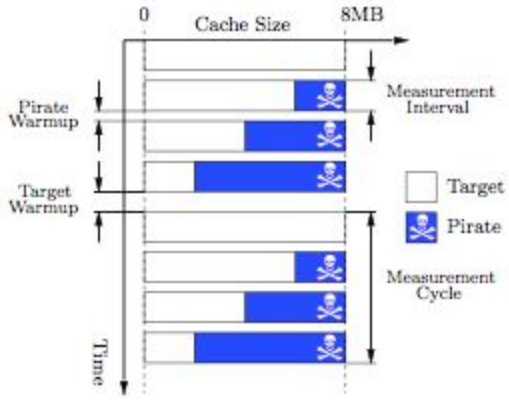
PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.

- Advantage
    - Can be used to get hardware counters for cache misses

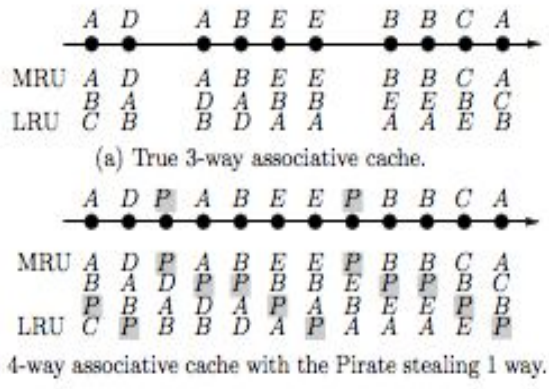
5. Cache Pirate

It's a low-overhead method for accurately measuring application performance (CPI) and off-chip bandwidth (GB/s) as a function of its the available shared cache capacity, on real hardware, with no modifications to the application or operating system.

- Disadvantages
    - Its goal is to steal cache without impacting target performance. Hence it doesn't steal bandwidth



    - Time is measured in interval and is not continuous
    - Dynamic cache size change: shouldn't pause
        - decrease cache allocation: pause the target; warm up pirate's cache
        - increase cache allocation: pause the pirate; warm up target's cache
    - Impact of kernel's time-sliced process scheduler is unknown
- Learnings
    - Zero fetch ratio: required data of pirate is in cache



(a) True 3-way associative cache.

(b) 4-way associative cache with the Pirate stealing 1 way.

- Sequential array access pattern
- Threads to make sure entire cache is in memory
- Core affinity: Target and pirate should be placed on separate cores

6. **StressMark**

   Similar to Cache pirate, used to steal cache from a co-run target application

   - Disadvantage
     - The miss ratio of the stress application is determined by its interaction with the target application's cache footprint. As this behavior varies during execution, they are only able to determine the average cache utilization for a given execution. This presents a problem for applications where varying phases have distinct cache behavior as the minimum and maximum cache used by the target application could vary significantly from the average. Unfortunately they do not present sufficient evaluation of their approach to determine if this is significant.

Work Done

Brief Overview:

1. Development of the CBpirate tool to help empirical study of the performance of the aforementioned algorithms further. This tool gives the users the option to specify various parameters which alter the execution environment at runtime.
2. We have also implemented and derived experimental results from the execution of both the iterative and recursive implementations of the following algorithms on CBpirate:

Algorithm	**Depiction in Result Table and Source Files**
Matrix Multiplication	Mat_mul_ikj_ipara, mat_mul_para_rec
Parenthesis Problem	Parenthesis_iter, parenthesis_rec
Gap Problem	Gap_iter, gap_rec
Floyd Warshall's All Pairs Shortest Path Algorithm.	Fwapsp_iter, fwapsp_rec

Parallel Randomized Connected Components	Det_par_cc, rand_par_cc

3. A major improvement offered by CBpirate over Cache Pirate is that it supports the execution of multithreaded applications and is itself multithreaded as well. Also, it can limit the number of cores the application/program can use, to maintain complete control over the availability of the cache to the running programs. It is however recommended that the core on which CBpirate is running, must not be used by any other application.

Algorithm:

1. The CBpirate steals requested cache amount for the time interval(configurable by user) and requested memory-cache bandwidth within the time interval. The user will provide cache amount and memory-cache bandwidth to be stolen for each time interval. The first entry in the configuration file will be for first time interval, second for the second and so on. If number of such entries is less than total number of intervals of time for which target is executing, then the entry, corresponding to cache amount and memory-cache bandwidth to be stolen, is chosen in round robin fashion.
2. Depending upon speed of processor and execution time of program and cache access latency of the program, the sampling time interval should be set.
3. During each time interval, CBpirate reads the amount of cache(unit: number of ways of cache) to be stolen from the config file. The recommendation is to steal maximum 50% of cache i.e. half the number of ways of cache if CBpirate is provided 50% of the cores.
4. CBpirate does sequential access and cache blocks from all the cache lines are touched. Stealing all the lines of cache is essential since it is not known which cache line will be used by target.
5. CBpirate uses huge pages so that sequential access along all the lines of cache for a particular way of cache is guaranteed even if the processor uses virtual addressing and cache is physically indexed. The system on which CBpirate runs should be configured to have a hugepage of size greater than size of a way of cache.
6. The number of huge pages accessed depends on amount of cache to be stolen. E.g. If half of the cache is to be stolen and cache is 12 way associative, then 6 huge pages are accessed. This will make sure that 6 ways of cache are stolen.
7. CBpirate outputs the number of cache misses, cycles per instruction and other parameters for each time interval. If the fetch ratio is high or cycles per instruction is high, number of CBpirate threads should be increased & rerun via command line because the cache amount that was supposed to be stolen isn't. The rate at which the cache blocks are touched by CBpirate will be increased as the number of threads increases and this causes the cache blocks to be kept hot and so the required amount of cache is stolen from the target.
8. To create memory-cache block transfers i.e. to change the apparent bandwidth, the idea is to update memory of such a address location that content of cache line of only a fixed index is changed i.e. old content of fixed index is flushed out and new content is brought into the cache line. Even if the target performs less cache accesses as compared to CBpirate, the maximum cache stolen in order to change the bandwidth would be equal to block size * no of ways. This is

an acceptable error. CBpirate uses index zero, when changing the bandwidth i.e. it updates first byte of hugepage. CBpirate maintains a list of hugepages equal to twice the number of ways of cache and a pointer to the hugepage in the list that was used last time to create memory-cache block transfers. Whenever there is a request to create memory-cache block transfer, the pointer to the hugepage in the list is updated in the round robin fashion and first byte of that hugepage is updated.

NOTE:

1. The cache replacement policy is considered to be Least Recently Used(LRU). The algorithm works fine even if the cache replacement policy is pseudo-LRU. When all the bits are unset during setting of last LRU bit, cache misses may increase since the target may replace cache blocks of CBpirate but this is an acceptable error.
2. The cache that is stolen should be shared by CBpirate and target i.e. CBpirate and target should share the cache. So its the LLC(Last level cache) from where CBpirate steals cache amount and effective bandwidth.

Experimental results

System used

Linux Fedora:	4.7.10-100.fc23.x86_64
L3 No of ways:	12
L3 No of lines:	4096
L3 total size:	3 MB
L3 shared cpu list:	0-3
Hugepage size:	2 MB
All cpu list:	0-3
Threads / core:	2
Cores / socket:	2
Sockets:	1
/proc/cmdline:	BOOT_IMAGE=/vmlinuz-4.7.10-100.fc23.x86_64 root=/dev/mapper/fedora-root ro rd.lvm.lv=fedora/root rd.lvm.lv=fedora/swap rhgb quiet LANG=en_US.UTF-8 isolcpus=2,3

Other settings:

- Add the kernel parameter "isolcpus=<CPU_ID>" to the boot loader to make sure no other process is scheduled to these CPUs
- Set core affinity for pirate to second half of all processors (should be same as CPU_ID provided to isolcpus)
    - Taskset -cp <corelist> <pirate's pid>
- Set core affinity for target to first half of all processors
    - Taskset -cp <corelist> <target's pid>
- Pirate is given dedicated cores to make sure it steal caches

Configuration:

- Cache mentioned is no of ways to be stolen i.e. if the table has entry as one in the table under column “Cache”, the amount of cache to be stolen is equal to way size. The way size is total size of cache/ number of ways.
- BW mentioned is no of block transfers apart from normal stealing of cache

Table 1:

Cache	**BW**
1	0
1	0
2	0
2	0
2	0
2	36
2	0
2	0
2	8
2	0
2	120
0	0

Table 2:

Cache	**BW**
1	0
1	12
2	0

2	12
3	0
4	12
5	0
5	12
6	0
2	12
2	0
0	0

Actual configs:
Config 1:
Table1, No of threads=4, No of cores=1, time interval = 1s

Config 2:
Table2, No of threads=4, No of cores=1, time interval = 1s

Config 3:
Table2, No of threads=4, No of cores=2, time interval = 1s

Config 4:
Table1, No of threads=4, No of cores=1, time interval = 100 ms

Config 5:
Table2, No of threads=4, No of cores=2, time interval = 100 ms

Config 6:
same as Config1; interval = 400ms

Config 7:
same as Config2; interval = 400ms

Programs	**without cbpirate**	**with cbpirate (config1)**	**with cbpirate (config2)**	**With cbpirate**

				(config3)
mat_mul_ikj_ipara (2048)	344.147	511.388	482.884	826.329
mat_mul_para_rec (2048)	407.745	585.393	567.955	962.457
parenthesis_iter (2048)	125.818	179.383	176.628	266.84
parenthesis_rec (2048)	97.8242	134.734	136.301	212.545
gap_iter	15.1171	34.1887	NA	35.9738
gap_rec	8.59868	17.7021	NA	18.0574
fwapsp_iter	9.27599	20.097	NA	20.0804
fwapsp_rec	9.68807	19.7396	NA	19.2739

NOTE: NA mentioned stands for Not available. We didn't run the remaining algorithms with the config2. With Config2, we are unable to steal the required amount of cache since the number of cores are too less and CBpirate is not successful in stealing the requested amount of cache and bandwidth. This configuration was used to only proof that if the number of threads or number of cores are less i.e. the rate at which CBpirate touches the cache to steal the requested amount is less, than the requested amount of cache won't be stolen from the target. Setting of the number of threads should be done according to the speed of processors and the maximum amount of cache to be stolen.

Programs	**without cbpirate**	**with cbpirate (config4)**	**with cbpirate (config5)**	**with cbpirate (config6)**	**with cbpirate (config7)**
det_par_cc (ca-AstroPh)	0.908771	1.97852	1.77013	1.80838	1.89085
rand_par_cc (ca-AstroPh)	1.63208	2.77474	3.06013	3.18897	3.2864

Observations:

- From the execution timings collected, we have made the observation that, on running the programs normally and along with CBpirate, we see that there is considerable difference between the timings observed for both the cases. The execution time significantly increases for the case when the algorithms are executed along with CBpirate.
- As and when the cache & effective bandwidth to be stolen is specified, the execution time of the algorithm increases.
- We can also observe that, among the recursive and iterative implementations of the concerned algorithms, the recursive implementations have lesser percentage difference when the cache/bandwidth stolen is increased.
- For some cases the time of execution is not increasing in the ratio it should. There could be two reasons for that:
    - Algorithm is cache adaptive
    - CBpirate didn't succeed in stealing the requested cache and effective bandwidth. We believe that this is true only for programs that have very small time of execution. Looking at other parameters like fetch ratio, miss ratio on a per core basis will give us more info here.

Conclusion

The time of execution of all the programs that are run without the CBpirate are less than compared to time of execution of same programs that are run with CBpirate. Thus, CBpirate is successfully stealing the cache & bandwidth as specified and thus can be used for further empirical study of the behavior of cache oblivious algorithms.

Future Work

CBpirate

The current implementation of CBpirate assumes that it is running on two cores numbered two and three. The implementation should be changed so that CBpirate takes corelist on which it is supposed to run and then set the core affinities of threads that steal cache and bandwidth accordingly

Experiments

- The experiments are on a desktop machine which is six years old. Two processors are virtual(Hyper threading) and all processors are slow as compared to present day processors. We feel that due to high cache access latency and low processor speed, the cache supposed to be stolen by CBpirate wasn't. Running on server machine with a high speed processor with physical cores and low cache access latency will guarantee that required cache is stolen by CBpirate and we may get better results of experiments.

- Testing of CBpirate was done on basis of commands per instruction (CPI). We need to test CBpirate on basis of fetch ratio for confirmation. This can be done on Nehalem processors.
- Run experiments with varied configuration of cache amount and memory-cache bandwidth to be stolen.

References:

[1] M. Bender, R. Ebrahimi, J. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley, "Cache-adaptive algorithms," Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), 2014.

[2] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. Kuszmaul, C. Leiserson, A. Solar-Lezama, and Y. Tang, "AUTOGEN: Automatic Discovery of Cache-Oblivious Parallel Recursive Algorithms for Solving Dynamic Programs," Proceedings of the 21st SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2016.

[3] P. Ganapathi, R. Chowdhury, Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems, PhD Dissertation.

[4]http://erikdemaine.org/theses/alincoln.pdf

[5]http://www3.cs.stonybrook.edu/~rebrahimi/papers/SODA-talk-Roozbeh.pdf

[6]https://it.uu.se/research/publications/reports/2011-001/2011-001-nc.pdf

[7]https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html

[8] https://web.eecs.umich.edu/~zmao/Papers/xu10mar.pdf
[9] https://it.uu.se/research/publications/reports/2011-001/2011-001-nc.pdf
[10] http://icl.utk.edu/papi/
[11] https://github.com/RRZE-HPC/likwid
[12] http://valgrind.org/docs/manual/cg-manual.html