Detailed Characterization of HPC Applications for Co-Scheduling

Josef Weidendorfer
Department of Informatics, Chair for Computer Architecture
Technische Universität München
weidendo@in.tum.de

Jens Breitbart
Department of Informatics, Chair for Computer Architecture
Technische Universität München
j.breitbart@tum.de

ABSTRACT

In recent years the cost for power consumption in HPC systems has become a relevant factor. One approach to improve energy efficiency without changing applications is co-scheduling, that is, more than one job is executed simultaneously on the same nodes of a system. If co-scheduled applications use different resource types, improved efficiency can be expected. However, applications may also slow-down each other when sharing resources. With threads from different applications running on individual cores of the same multi-core processors, any influence mainly is due to sharing the memory hierarchy. In this paper, we propose a simple approach for assessing the memory access characteristics of an application which allows estimating the mutual influence with other co-scheduled applications. Further, we compare this with the stack reuse distance, another metric to characterize memory access behavior.

Keywords

Co-Scheduling; Memory Hierarchy; Application Characterization

1. INTRODUCTION

Improvements of computational power of HPC systems rely on two aspects: On the one hand, increased performance comes from an increased number of nodes[1]. On the other hand, node performance itself is expected to grow with newer hardware. Since quite some time, the latter can only be achieved by more sophisticated node designs. General purpose CPUs consist of an increasing number of cores with complex multi-level cache hierarchies. Further, the need for better power efficiency results in increased use of accelerators. Memory modules either have separate address spaces (accelerator vs. host) or are connected in a cache-coherent but non-uniform memory-access (NUMA) fashion. However, the latter makes it even more difficult to come up with high-performance codes as it hides the need for access locality in programs. All these effects result in a productivity problem for HPC programmers: development of efficient code needs huge amounts of time which is often not available. In the end, the large complex HPC systems may have a nice theoretical performance, but most real-world codes are only able to make use of just a small fraction of this performance.

Co-scheduling is a concept which can help in this scenario. We observe that different codes typically use (and are bound by) different kinds of resources during program execution, such as computational power, memory access speed (both bandwidth or latency), or I/O. However, batch schedulers for HPC systems nowadays provide dedicated nodes to jobs. It is beneficial to run multiple applications at the same time on the same nodes, as long as they ask for different resources. The result is an increased efficiency of the entire HPC system, even though individual application performance may be reduced.

To this end, the scheduler needs to know the type of resource consumption of applications to be able to come up with good co-scheduling decisions. HPC programs typically make explicit use of execution resources by using a given number of processes and threads, as provided by job scripts. Thus, we assume that co-scheduling will give out dedicated execution resources (CPU cores) to jobs. However, cores in the same node and even on the same multi-core chip may be given to different applications. In this context, co-scheduling must be aware of the use of shared resources between applications.

In this paper, we look at different ways to characterize the usage of the memory hierarchy by applications. Especially, the results should help in predicting mutual influence of applications when running simultaneously on the same node of an HPC system. First, we look at so-called stack reuse histograms. These histograms provide information on exploitation of cache levels and main memory. However, being an architecture-independent metric without time relation, they cannot include information about how much of a hardware resource is used. However, this is crucial to understand whether an execution gets slowed down when another application shares resources. Thus, secondly, we propose the explicit measurement of slowdown effects by running against a micro-benchmark accessing with a given memory footprint at highest access rate. We compare slowdown results with the reuse histograms of two specific real-world applications.

[1]A node is one endpoint in the network topology of an HPC system. It consists of general purpose processors with access to shared memory and runs one OS instance. Optionally, a node may be equipped with accelerators such as GPUs).

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286951

One is an example use-case of the numerical library LAMA [11], consisting of a conjugate gradient (CG) solver kernel. The other is MPIBlast [12], an application from bioinformatics, used to search for gene sequences in an organism database.

We plan to use the prediction for co-scheduling decisions. Within this larger context, an extension of the batch scheduling system[2] is planned which also takes online measurement from node agents into account (the node agent actually is an extension of `autopin` [10], a tool to find best thread-core bindings for multi-core architectures). The vision is that the system should be able to train itself and learn from the effects of its co-scheduling decisions. If there is no characterization of an application yet, the node agent could use the proposed micro-benchmark to gradually get more knowledge about the application for better co-scheduling. Use of the micro-benchmark with its known behavior allows us to obtain such knowledge which depends only on the application and node architecture. This is in contrast to observing slowdowns of arbitrarily running applications.

To get the reuse distance histograms of applications, we developed a tool based on Pin [13]. This tool allows the observation of the execution of binary code with the help of dynamic runtime instrumentation. This way, HPC codes with complex dependencies (in our case Intel MPI, Intel MKL, and Boost) can easily be analyzed without recompilation. Thus, the main contributions of this paper is detailed analysis of the relation of reuse distance histograms and slowdown behavior of applications triggered by a co-running micro-benchmark with one given reuse-distance.

In the next section, we present measurements about co-scheduling scenarios for the applications analyzed, providing motivation and a reference for our discussion. Then we shortly describe our Pin-based tool and the micro-benchmark. Finaly we present detailed slowdown figures for both the applications and the micro-benchmark itself.

2. CO-SCHEDULING MEASUREMENTS

As reference for later discussion, we first present some measurements of co-scheduling scenarios with MPIBlast and LAMA, as already shown in [3]. The system used is equipped with two Intel Xeon E5-2670 CPUs, which are based on Intel's Sandy Bridge architecture. Each CPU has 8 cores, resulting in a total of 16 CPU cores in the entire system. Both L1 (32kB) and L2 (256kB) caches are private per core, the L3 cache (20MB) is shared among all cores on a CPU socket. The base frequency of the CPU is 2.6GHz. However, "Turboboost" is enabled (i. e., the CPU typically changes the frequency of its cores based on the load of the system). However, we do not use Hyperthreading. All later measurements in this paper also were carried out on this system.

For both applications, typical input data is used which results in roughly the same runtimes if executed exclusively. Fig. 1 shows performance and efficiency of various co-scheduling scenarios. From the 16 cores available, the X axis shows the number of cores given to MPIBlast. The remaining cores are assigned to LAMA, respectively. Threads are alternatingly pinned to CPU sockets: e.g. for the scenario with 4 LAMA threads, two are running on each socket. The efficiency is given relative to the best dedicated runtimes of the applications. Note that MPIBlast must be run with at least 3 processes[3]. The best co-scheduling scenario (defined as highest combined efficiency of around 1.2) is with 11 cores given to MPIBlast and 5 cores to LAMA. This shows that LAMA and MPIBlast can benefit from being co-scheduled. In the referenced paper, we also showed energy consumption benefits are even higher. Section 5 will provide insights into why the positive effects are possible.

3. REUSE DISTANCE HISTOGRAMS

For the effective use of caches, good temporal locality of memory accesses is an important property of any application. Temporal locality exists when a program accesses memory cells multiple times during execution. If such accesses are cached, following accesses to the same location can be served much faster, speeding up execution.

To better understand how efficiently an application can exploit caches, a precise definition of temporal locality for a stream of memory accesses is helpful. The *Stack Reuse Distance*, introduced in [1], is the distance to the previous access to the same memory cell, measured in the number of distinct memory cells accessed in between[4] (for the first access to an address, the distance is infinity). For a fully associative cache of size S with least-recently-used (LRU) replacement, a memory access is a cache hit if and only if its stack reuse distance is lower than or equal to S. Thus, if we generate a histogram of distances of all memory accesses from the execution of a program, we immediately can see from this histogram how many of these accesses will be cache hits for any given cache size: looking at the area below the histogram curve, this is the ratio of the area left to the distance signifying the cache size in relation to the whole area. Because the behavior of large processor caches (such as L3) is similar to the ideal cache used in the definition above, the histogram of stack reuse distances is valuable for understanding the usage of the memory hierarchy by a sequential program execution.

Figures 2 and 3 show histogram examples for sequential runs of the applications analyzed in this paper. Many accesses at a given distance means that a cache covering this distance will make the accesses cache hits. Looking e.g. at the three histograms in Fig. 2, where we marked the L3 cache size, it is obvious that even for a small run with 500^2 unknowns, for a large portion of accesses, LAMA has to go to main memory.

The histogram cannot directly be measured with hardware support. In the following, we shortly describe our own tool able to get reuse distance histograms. It maintains an exact histogram taking each access into account. Due to that, the runtime of applications is on average around 80 times longer compared to native execution.

[2]Here we will look at Slurm.

[3]MPIBlast uses a two level master-worker scheme with one process being a "supermaster" and at least one other process being a master. Both supermaster and master distribute work to at least one worker.

[4]Papers from architecture research sometimes define the term *reuse distance* of an access from the previous access to the same memory cell as being the number of accesses in-between. This gives a time-related distance different to our definition here. A reuse distance in this paper is always the stack reuse distance.

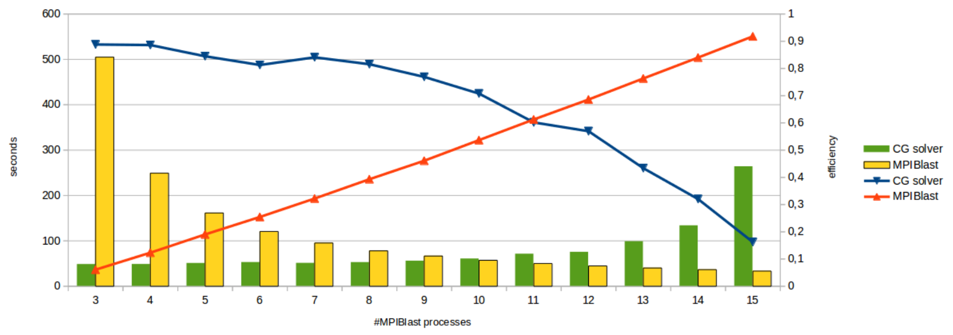


Figure 1: Runtimes and efficiency of co-scheduling scenarios. The core count used by MPIBlast is given on the X axis, other cores are used for LAMA/CG solver (from [3]).

PinDist: A Tool for Deriving Reuse Distances

With the use of Pin [13] we developed a tool that is able to observe all memory accesses from the execution of a program to obtain its reuse distance histogram. For this, Pin is dynamically rewriting binary code in memory directly before execution similar to just-in-time (JIT) compilers. Our tool maintains a stack of accesses to distinct memory blocks of size 64 bytes according to recency of accesses. For each access observed, the corresponding entry for the accessed block is moved to the top of the stack, and the depth where the block was found — this is the distance of the access — is aggregated in the histogram (on first access, we create a new entry using distance infinity). More precisely, we use distance buckets, allowing for a faster algorithm as given in [9]. As suggested in [15], we ignore stack accesses which can be identified at instrumentation time.

PinDist is available on GitHub[5].

4. DISTGEN: CONTROLLED MEMORY ACCESS BEHAVIOR

DistGen is a micro-benchmark written to produce executions exposing given stack reuse distances (and combinations). For this, it reads the first bytes of 64-byte memory blocks in a given sequence as fast as possible. For example, to create two distances, it allocates a memory block with the size of the larger distance. The smaller distance is created by accessing only a subset of the larger block, containing the required number of memory blocks corresponding to the smaller distance. Depending on the required distance access ratio, the smaller and larger blocks are alternately accessed. The expected behavior easily can be verified with our PinDist tool.

DistGen can run multi-threaded, replicating its behavior in each thread. It can be asked to perform either streaming access or pseudo-random access, prohibiting stream prefetchers to kick in. The later is not used in this paper, as it reduces the pressure on the memory hierarchy and does not provide any further information for our analysis. Further, it can be configured to either do all accesses independent from each other, or via linked-list traversal. The latter enforces data dependencies between memory accesses, which allows measuring worst-case latencies to memory. In regular intervals, DistGen prints out the achieved bandwidth, combined across all threads. DistGen is provided in the same GitHub repository as PinDist.

In the following, we used DistGen to simulate a co-running application with a given, simple memory access behavior: streamed access as fast as possible using exactly one specified reuse distance. To ensure occupying available memory bandwidth, we run DistGen on one CPU socket with four threads.

5. RESULTS

First, we analyzed LAMA and MPIBlast by extracting the reuse distance histogram from typical executions. In all histograms, we marked L2 and L3 sizes. All accesses with distances smaller than L2 size are not expected to influence other cores as L1 and L2 are private. In the range between L2 and L3 size, co-running applications may compete for cache space, and due to benefiting from reuse (accesses in this range are hits due to L3), slowdowns are expected if data is evicted by the co-running application. All accesses with distances larger than L3 size go to main memory, and need bandwidth resources which also is shared by all cores on a CPU, and thus a potentially another reason for slowdown.

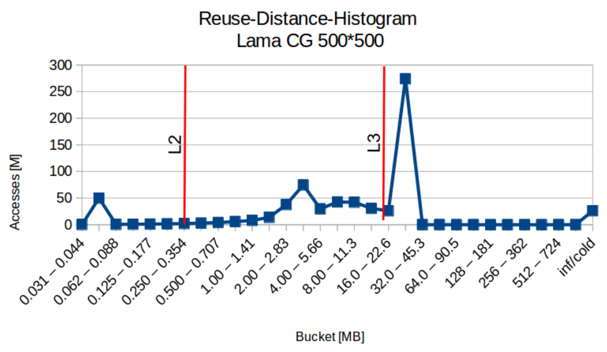
Reuse Distances

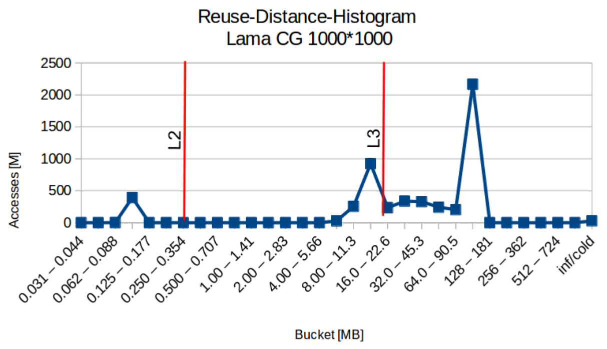
The CG solver from LAMA is running sequentially in 3 configurations solving a system of equations with different number of unknown. Figure 2 shows the resulting histograms with markers for L2 and L3 cache sizes. It is interesting to observe spikes with heavy access activity at 3 distances which move upwards in the same fashion with higher number of unknowns. The solver does a sparse-matrix vector operation and multiple vector operations per iteration.

1. The large distance corresponds to the total size of the matrix in memory in CSR (compressed sparse row) format.
2. The middle spike corresponds to the vector length (e.g. 8 million doubles for the 1000^2 case).
3. The lower spike comes from re-referencing parts of the vector in the SpMV operation due to the sparsity structure (the example uses a 2D 5-point stencil).

In all LAMA cases, stream-accessing the sparse matrix cannot exploit caches and results in heavy memory access needs. From measurements we observed that LAMA performance does not improve beyond 8 threads with dedicated hardware. The reason is that 4 LAMA threads on each

[5]`https://github.com/lrr-tum/reuse`





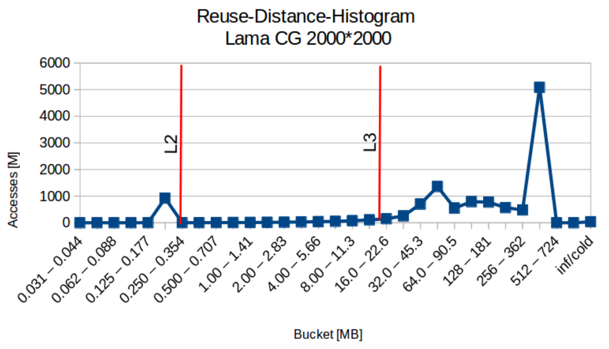
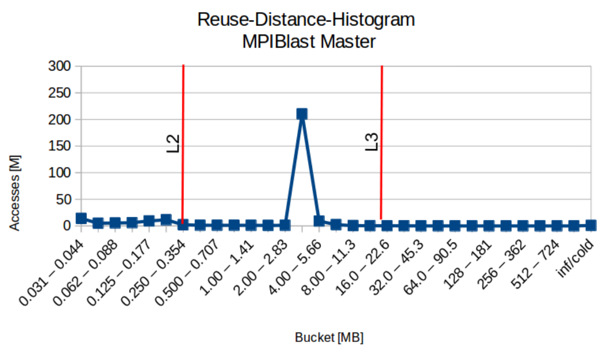


Figure 2: Reuse Distance Histogram for LAMA with 500^2 (top), 1000^2 (middle), and 2000^2 (bottom) unknowns.



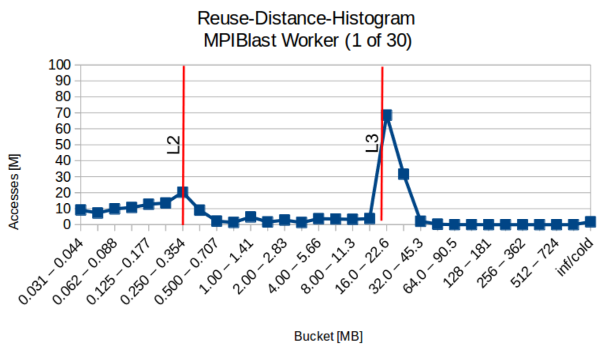


Figure 3: Reuse Distance Histogram for MPIBlast. Master (top) is distributing chunks of workload to workers (bottom) which all have the same histogram.

CPU socket obviously saturate the available bandwidth to memory. Thus, LAMA performance is memory bound at that point.

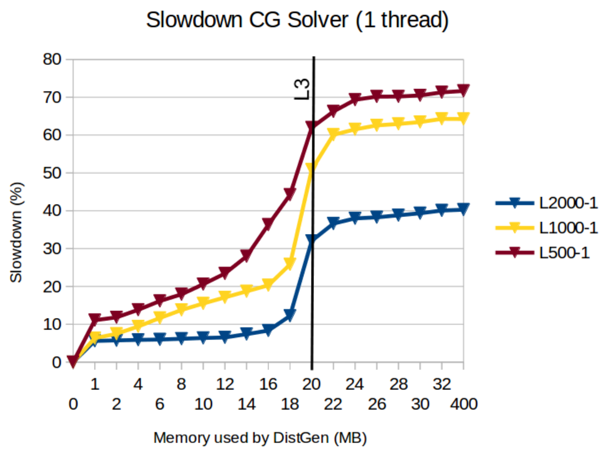
Figure 3 shows two histograms of one MPIBlast run. The master MPI-task only has one spike, most probably corresponding to buffer sizes for reading input data and distributing data to workers. The supermaster process is doing nothing in this configuration and therefor not shown. Further, we show the histogram of an MPI worker (we run this with 32 MPI tasks in total, resulting in 30 worker processes). Histogram of all workers are similar. Apart from quite some activity below and around L2 cache size, there is a spike at the distance of L3 size. However, from measurements, we did not really see much traffic to main memory. The solution to this mystery lies in the fact that we do not show the number of accesses for the bucket with the lowest distances (from 0 to 31K). For almost all reuse histograms shown, the spike “skyrockets” the visualized range. E.g. for each MPIBlast workers, it is more than 15 billion accesses.

This shows a drawback not only of our visualization, but for such histograms in general. Even if 100% of accesses were represented by area in the histogram, we cannot see the frequency of accesses. The histogram may hint at memory boundedness, but the frequency of accesses may be so low that the hint is misleading. For more details, we have to look at real measurements showing the influence in co-running applications.

Slowdown Behavior

To predict the slowdown of applications being co-scheduled, we need to co-run it with a benchmark for which we know the behavior. Measurements are done using one CPU socket. DistGen is always running with 4 threads. That is, the 1 MB memory usage point in the figures actually means that each thread is traversing over its own 256 kB. While in this point mostly private L2 is used by DistGen, due to the strict inclusiveness property of the L3 cache in Intel processors, this still requires 1 MB space from L3.

Figure 4 shows the performance of the LAMA CG solver while being co-scheduled with DistGen with different distances. The reuse distance histograms predicted that the CG solver for both 500^2 and 1000^2 unknowns partially use L3 cache, whereas with 2000^2 unknowns there is hardly any benefit for L3 accesses. This can be seen clearly in Fig. 4. The performance with 2000^2 unknown gets severely reduced once DistGen starts consuming main memory bandwidth, whereas with 500^2 and 1000^2 unknowns we already see a performance degeneration when DistGen starts to consume L3 cache. Furthermore, the maximum overall performance hit is higher with 500^2 and 1000^2 unknowns as they benefited



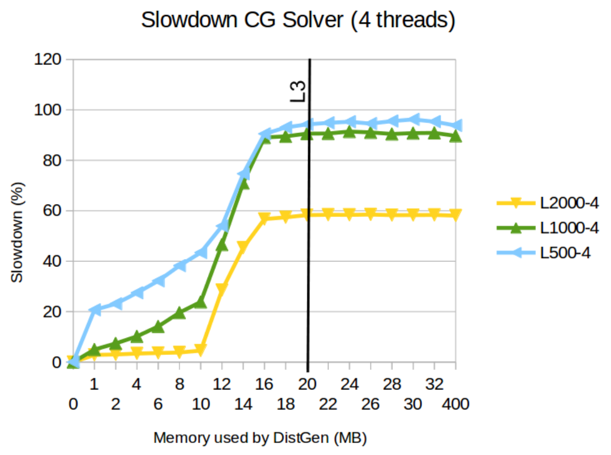


Figure 4: Slowdowns perceived by LAMA with 1 (top) and 4 threads (bottom), running against DistGen with 4 threads.

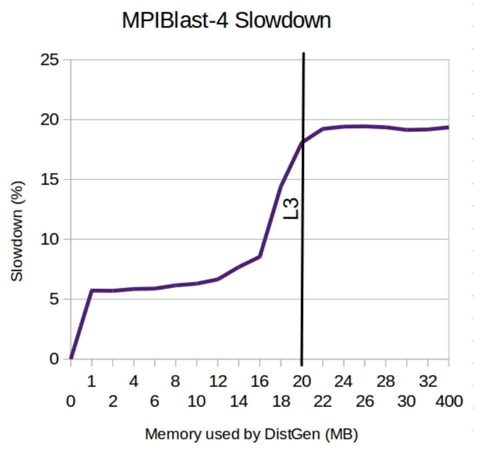


Figure 5: Slowdowns perceived by MPIBlast with 4 threads, running against DistGen with 4 threads.

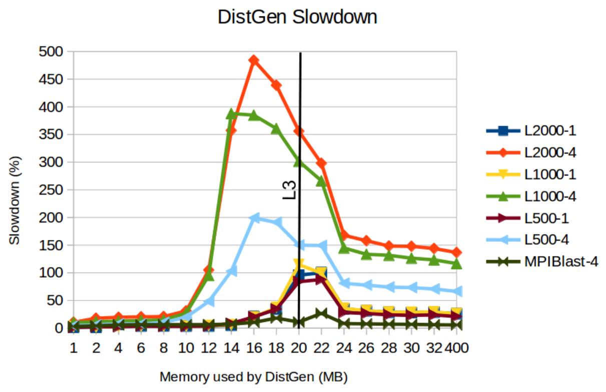


Figure 6: Slowdowns perceived by our micro-benchmark DistGen when running against various applications.

from L3 cache. The maximum overall performance hit is higher when using 4 threads compared to 1 one thread. This results from the fact that a single thread cannot consume the whole bandwidth provide by a CPU socket, whereas 4 threads can. Interestingly, the maximum slowdown with four CG solver threads is already reached with 16 MB DistGen usage. This shows the mutual influence between applications. We attribute this to the CG solver threads evicting cache lines from DistGen such that DistGen starts to use main memory bandwidth.

Figure 5 shows the performance of four MPIBlast processes when being co-run with DistGen with different distances. Again, the results of this measurement closely resembles the ones shown in the reuse distance histogram. MPIBlast mostly relies on its private L2 cache and therefore hardly reacts to DistGen consuming L3 cache. Once DistGen consumes main memory bandwidth we see a slowdown of MPIBlast, as it was predicted by the reuse distance histogram. We assume the initial 5% performance hit of MPIBlast when being co-run with DistGen to be the result of reduced CPU clock frequency. With four idle cores Intels Turboboost can notably increase the CPU clock frequency. But when DistGen is running, all 8 cores are active and the CPU temperature is increased leaving less opportunities to increase the CPU frequency. Overall, the maximum performance hit of MPIBlast ($\leq 20\%$) is far lower than that of the CG solver ($\geq 90\%$). We cannot obtain this information from the reuse distance histograms.

Figure 6 shows the performance of DistGen when being co-run with the various applications. We can gather almost the same information from these figures as we did from the previous ones, but our tool reacts much more apparent (up to 500%). All variations of the CG solver slow down DistGen when it uses main memory bandwidth, whereas MPIBlast hardly results in a slowdown. The single threaded CG solver requires less resources compared to the versions using 4 threads, where the slowdown perceived by DistGen peaks already at 16 MB. This confirms our assumption from above that DistGen is forced to go to main memory at this point.

Overall, we observe that the performance of DistGen when being co-run with an unknown application can provide valuable insights into the other application. Such information will definitely be useful to automatically determine if applications benefit from co-scheduling.

6. RELATED WORK

The stack reuse distance histogram has shown to be very helpful in analysing memory usage and hinting at tuning opportunities [2]. There are quite some papers suggesting fast methods for its determination in software-based tools, as exact measurement is impossible using hardware. However, authors of [6] propose a statistical approximation using hardware measurements which is extended to multi-cores in [16]. We note that these methods, being statistical, only work well with regular memory usage patterns. None of the papers use the reuse distance histogram in the context of analyzing co-scheduling behavior.

Characterizing co-schedule behavior of applications by measuring their slowdown against micro-benchmarks is proposed by different works. MemGen [5] is focussing on memory bandwidth usage, similar to Bandwidth Bandit [7] which is making sure not to additionally consume L3 space. Bubble-Up [14] is very similar to our approach in accessing memory blocks of increasing size. However, we vary the number of threads co-run against our benchmark.

7. CONCLUSION

In this paper, we studied various ways of a-priori analysis of applications for suitability to improve system throughput via co-scheduling. Reuse distance histograms combined with slowdown measurements proved very useful in this context. We will use these methods in a modified job scheduler.

To avoid slowdown effects of co-running applications on the same multi-core CPU, recent hardware (some versions of Intel Haswell-EP CPUs) allows to configure L3 cache partitions for use by subsets of cores on the chip [8]. Instead of avoiding specific co-schedulings, one can dynamically configure resource isolation to avoid slowdown effects. In [4] it was shown that this can be helpful. We will extend our research in this direction.

Acknowledgments

The work presented in this paper was funded by the German Ministry of Education and Science as part of the FAST project (funding code 01IH11007A).

8. REFERENCES

[1] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19:353–357, 1975.

[2] K. Beyls and E. D'Hollander. Platform-independent cache optimization by pinpointing low-locality reuse. In *Proceedings of International Conference on Computational Science*, volume 3, pages 463–470, June 2004.

[3] J. Breitbart, J. Weidendorfer, and C. Trinitis. Case study on co-scheduling for hpc applications. In *International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS 2015)*, Beijing, China, 2015.

[4] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. *SIGARCH Comput. Archit. News*, 41(3):308–319, June 2013.

[5] A. de Blanche and T. Lundqvist. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing*, 71(4):1451–1483, 2015.

[6] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE Internat ional Symposium on*, pages 55–65, March 2010.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013.

[8] Intel. Improving real-time performance by utilizing cache allocation technology. white paper. White Paper, April 2015. Document Number: 331843-001US.

[9] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. *SIGMETRICS Perform. Eval. Rev.*, 19(1):212–213, Apr. 1991.

[10] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. autopin – automated optimization of thread-to-core pinning on multicore systems. In P. Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Computer Science*, pages 219–235. Springer Berlin Heidelberg, 2011.

[11] J. Kraus, M. Förster, T. Brandes, and T. Soddemann. Using LAMA for efficient amg on hybrid clusters. *Computer Science-Research and Development*, 28(2-3):211–220, 2013.

[12] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, and W.-c. Feng. Massively parallel genomic sequence search on the Blue Gene/P architecture. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[14] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.

[15] M. Pericas, K. Taura, and S. Matsuoka. Scalable analysis of multicore data reuse and sharing. In *Proceedings of the 28th ACM International Conference on Supercompu ting*, ICS '14, pages 353–362, New York, NY, USA, 2014. ACM.

[16] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.