So many performance events, so little time

Gerd Zellweger Denny Lin Timothy Roscoe
Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
gerd.zellweger@inf.ethz.ch dlin@student.ethz.ch troscoe@inf.ethz.ch

ABSTRACT

Many modern workloads are a heterogeneous mix of parallel applications running on machines with complex processors and memory systems. These workloads often interfere with each other by contending for the limited set of resources in a machine, leading to performance degradation.

In principle, one can use hardware performance counters to characterize the root causes of interference inside a modern machine. However, we show that current facilities in today's operating systems mean that such an analysis requires careful consideration of the intricacies of specific hardware counters, domain knowledge about the applications, and a deep understanding of the underlying hardware architecture.

In contrast, we present the design of Glatt, which is able to automatically identify the root causes of interference in a machine online. Glatt can expose this information to a runtime or the operating system for predicting better thread assignments or for dynamically increasing/decreasing parallelism within each runtime.

1. INTRODUCTION

Modern server workloads are often a mix of different applications, each with their own parallel runtime. Efficiently scheduling this mix is famously difficult on modern hardware for two reasons: First, today's complex cache hierarchies, bus topologies, and CPU architectures lead to surprising interactions that negatively impact the performance of individual applications. Secondly, many modern applications rely on runtime systems that make their own scheduling decisions, and each runtime's view is limited by what little information an operating system can provide. Consequently, such applications either assume they have the whole machine to themselves, or at best employ simple heuristics for dynamically choosing their degree of true parallelism.

APSys '16, August 4–5, 2016, Hong Kong, Hong Kong

 ISBN 978-1-4503-4265-0/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2967360.2967375

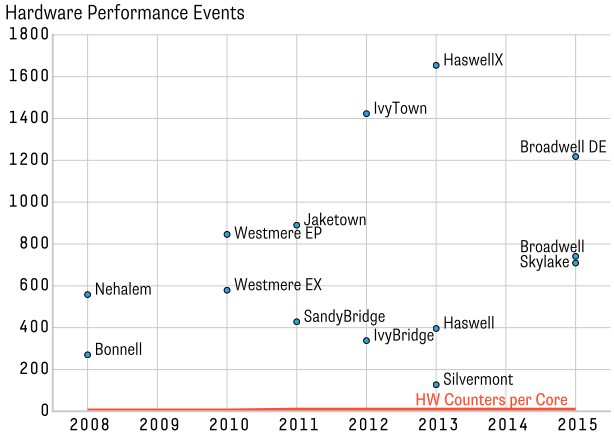


Figure 1: The number of hardware events (including subevents) available on different Intel architectures over time vs. the number of general-purpose hardware counters available for measuring events (8 per core on current Intel architectures).

In principle, performance counters provide knowledge regarding the state of the machine and can be used to detect workload interactions that cause inefficient execution. However, the *semantic gap* between counter values and observed performance is large, and the *sheer complexity* of hardware instrumentation available on a given machine makes choosing which measurements to take, and interpreting the resulting data, a daunting task which must be repeated for each new processor.

Figure 1 shows the number of monitorable events for several different Intel processors.[1] Note that current systems provide a very large number of possible events to monitor. The "right" set of events for diagnosing a given problem or predicting the performance of an application mix is difficult to find even for an expert, and the trend is getting worse.

We propose an alternative approach, and present the early design of Glatt, a service which uses hardware performance counters to detect application interference online and provides dynamic feedback to parallel runtime systems by recommending different thread placements or changes in the degree of requested parallelism. Glatt's novelty lies in the

[1]Numbers are generated from data files released by Intel, available at https://download.01.org/perfmon/.

fact that it automatically learns which combinations of hardware events on a given machine provide the best predictors of performance. This allows Glatt to run on a new machine out of the box, without needing manual tuning or expert knowledge.

The rest of the paper is organized as follows: In the next section (§2), we discuss the motivation for our work and survey related approaches. In Section 3, we show the challenges involved in building Glatt. Section 4 presents case studies of how performance events can be used to characterize interference. Finally, we discuss the design of Glatt (§5) and conclude in Section 6.

2. BACKGROUND

Hardware performance counters are available for all major hardware architectures and enabling them typically adds little overhead [7, 48] to the system. Therefore, using such counters to gain insights about the system is intriguing as it can be done online, without noticeably affecting the performance of applications.

Performance counters have been used in the past to address a wide range of problems: Detailed modeling of scientific applications [11,32,47], anomaly detection and problem diagnosis in production systems [7,8], and detection of security exploits [20]. Wang *et al.* showed that compilers can use performance counters to find optimal configurations by performing automatic profiling runs [43]. In addition, counters have found applications for minimizing energy consumption inside machines [30, 39].

Shared units such as caches, memory controllers, and SMT threads may negatively impact performance if threads that are executed together contend for those resources. A vast amount of research has been invested to develop contention-aware scheduling algorithms [51]. Such algorithms employ, for example, the LLC miss rate as a simple heuristic for contention [25, 50]. While cache contention can be addressed with recent technologies such as Intel's Cache Allocation Technology (CAT) [22], there is no corresponding solution for memory bandwidth partitioning. Such isolation measures can significantly increase the utilization of data centers [28]. However, they require complex online modeling in the OS to make an informed decision. Models relying on performance counters exist for predicting the cache size of applications or how threads will behave when sharing a cache [6, 13, 40]. Often those models rely on specific hardware events not available in every architecture [51]. This is a shortcoming which many of the presented approaches suffer from. Therefore, an OS needs to determine the model it uses based on the hardware it is running on. An alternative technique is to simply monitor the instructions per cycle (IPC) metric by considering data from previous runs to detect and terminate straggler processes in a data center [49]. Aside from caches, CPU features such as SMT can lead to performance improvements for some workloads but add few benefits for others [37]. Lo *et al.* avoid co-location of different workloads on the same core or SMT thread for reasons of interference [28].

All of the techniques described so far focus on the resource management and scheduling on an operating system level. However, the increasing parallelism inside machines and parallel programming models have led to a number of runtimes that do fine-grained scheduling of threads and memory management on their own: OpenMP [33] parallelizes `for` loops by dividing them into chunks of independent work items, which a set of OS-level threads then processes concurrently. The Cilk [18] and ForkJoin [34] programming models use task queues which are served by a set of OS-level threads. Virtual machines [31] and containers similarly need to make fine-grained scheduling decisions on their own. Such runtime schedulers can benefit from performance counter information to make informed thread placement decisions, or use the information to vary the parallelism dynamically at runtime. However, the current abstractions provided to user-level applications today typically target developers that want to do fine-grained performance analysis [10, 24, 27, 35, 46]. Such tools build on lower level abstractions [36, 44], but they are in turn cumbersome to program, do not abstract the hardware in a sufficient way, and are limited by the flexibility and number of counters provided by the hardware. Runtimes that could potentially benefit from such information would have to implement a large amount of additional logic in order to make sense of the data and gain any benefits. In addition, we argue that the performance monitoring should be done at a global, OS level in order to benefit from an omniscient view and be able to provide this information to various co-existing runtimes. We discuss the challenges involved in such a system in more detail in §3.

The mechanisms to exchange information between the OS and user-level runtimes have been around for a long time: Scheduler activations [3] use *upcalls* to inform the user-level runtime of processor reallocations and completed I/O requests. Tessellation [14] schedules cells (a set of CPU cores, memory pages, etc.) and uses a *Resource Allocation Broker* to distribute resources to cells and adjusts them online by using system-wide metrics such as cache misses and energy measurements and requesting progress reports from applications.

3. CHALLENGES

In this section, we first explain in more detail how hardware events are measured. We focus on Intel architectures, but the process is similar on AMD and ARM architectures. Then we discuss the challenges involved in building Glatt, an online hardware event monitoring system.

3.1 Programming performance counters

The interface for hardware performance counters on Intel's x86 machines is relatively straightforward. First, one has to choose an event from a set of predetermined microarchitectural events as defined in the Intel SDM [23]. Such events include, for example, cache misses on various levels, page walks, memory bandwidth utilization, and TLB misses. Next, a performance counter needs to be programmed to

measure the corresponding event. Recent Intel processors have three fixed counters (which can only measure specific events such as clock cycles and retired instructions) and eight programmable counters per core (four per SMT thread if SMT is enabled). The OS uses programmable counters to monitor a specific event by writing to model specific registers (MSR) with the correct event code and umask value. Afterwards, the OS can enable, disable, and reset counters by reading and writing MSR registers. Certain events can only be measured on a subset of the available counters. In addition, Intel supports a *Precise Event Based Sampling* (PEBS) facility which allows saving the CPU state (i.e., register values, addresses for a load/store, etc.) into a buffer in case a specific event occurs. PEBS is similar to *Lightweight Profiling* (LWP) in AMD processors. Linux provides a more generic interface for accessing performance counters in form of the `perf_event_open` [44] system call. It enables counters to be accessed through file descriptors and allows applications to control them with `ioctl` commands. In addition, it supports generic, architecture independent event descriptors for a small subset of the commonly available hardware events and provides the infrastructure to sample the counters at a given rate by writing values to a ring buffer.

In the remainder of this section, we cover certain limitations of performance counters which are currently not addressed in today's mainstream operating systems and explain how application runtimes could benefit from additional features.

3.2 What to measure?

In Figure 1, we observe that the number of measurable events is significantly higher than the number of available counters (16x for Silvermont SoC and 207x for HaswellX). Moreover, there is a high amount of variability even within the same microarchitecture. For example, the basic IvyBridge model defines only 338 events whereas the high-end IvyTown chips support 1423 events. In any case, the number of measurable events greatly outnumbers the number of available counters.

The figure raises two questions: First, from all the events, which ones should an operating system monitor to provide feedback to a runtime system, and secondly, how should this subset be multiplexed on the limited set of performance counters.

Event selection.

The problem of selecting the right subset of events arises from the fact that there are simply too many events. An OS does not know what subset is relevant for a given architecture or workload. To make matters worse, events differ between models and microarchitectures, let alone different vendors. Since understanding the various events is challenging even for an expert programmer simply optimizing code, it would be extremely difficult for an OS to generalize across all architectures. Although the domain knowledge about relevant events could potentially be encoded in a set of OS policies for a given architecture, such an approach does not scale. However, previous work has shown that certain events are highly correlated and successfully applied statistical procedures to filter out correlated events to reduce the set of 120 potentially measurable events to 34 in an ARMv7 chip [41].

Limited performance counters.

Even when only measuring linearly uncorrelated variables, the number of available counters may still not be sufficient. In the next section (§4), we will show how the detection of interference on different levels of caches and memory bandwidth alone already requires measuring of up to 10 events simultaneously. Techniques that multiplex counters to measure several events [5, 29] exist and are supported for example by PAPI [10]. Such techniques rely on statistical sampling by reprogramming a single counter with multiple different events during execution. MPX [29] analyzes the accuracy of such techniques and shows that in many cases, the introduced error is negligible.

3.3 Phase detection

Interactive applications, databases, and scientific simulations usually have different phases. For instance, consider a scientific workload that first loads data over the network or from disk, serializes the data into an in-memory data representation, and finally performs computations. While the first phase typically generates a large number of I/O requests and has low CPU utilization, the situation is reversed when data has been read into memory. Thread and memory allocation policies may vary according to phases. The problem is aggravated with interactive applications or databases where user-defined queries typically start a chain of events, which then results in different microarchitectural patterns. It makes sense for an interaction between an OS and the runtime (e.g., to decide on data or thread placement) to take place at the beginning of such a phase and not towards the end. An OS can detect such phases automatically by using existing phase detection techniques from signal processing [16] or by relying on information from runtimes about which threads belong together, when a certain task or query starts to be processed, and when a phase has completed.

3.4 Storing knowledge

None of the operating systems we are aware of currently monitor hardware events constantly by default. However, as samples can be obtained with low overhead, we propose an OS infrastructure which always enables the event measurement facility. In addition, gathered information should be stored in persistent storage, along with information about the machine state at any given point in time, e.g., thread placement. Past recordings should be retrievable by the OS and runtimes. With today's storage capacities, the aggregated data size should not pose a problem. For example, logging 8-byte values of 8 performance counters per core on a 64-core machine every second will result in a data rate of 4 KiB/s. Event logs may store data which is filtered and compressed. Such logs can in turn help predict the performance of future executions through regression analysis, or help detect anomalies due to interference and pass that information back to application runtimes. Furthermore, we

also envision such an infrastructure as a service for developers. It forms a basis for tools to detect anomalies in the form of bugs or performance issues (e.g., a sudden increase in cache miss rates caused by misaligned data structures) for subsequent versions of an application under development. Research operating systems have already explored the idea of a knowledge base and reasoning engine as part of the OS: Barrelfish [42] uses a *System Knowledge Base* (SKB) that stores information about the machine (including cores, NUMA regions, physical memory ranges, and device information). The information is in part encoded statically as hard-coded facts along with dynamically gathered information at boot and runtime and can be queried by applications and OS services [38].

3.5 Data analysis

In order to make sense of the measured data on-the-fly, an OS will have to apply well known techniques for dimensionality reduction, classification, regression modeling, and anomaly detection. Recent advances in machine learning have given rise to many highly optimized and open source software libraries [4,12,19]. They normally include efficient algorithms for the previously mentioned problems. Computations are often sped up by making use of accelerators such as GPUs. As part of this work, we plan to evaluate which algorithms work best in our context and how such libraries can be integrated in an operating system. While the number of events/features measured using performance counters is relatively small compared to traditional machine learning problem inputs, latency is critical in an operating system for short-lived jobs. Also, Glatt's computations should ideally happen in the background without interfering with applications.

3.6 Application feedback

Finally, Glatt should be able to exploit the available data and give useful feedback to multiple runtime systems running concurrently on a machine. The challenge lies in how such information should be presented such that it can actually be leveraged by existing runtime systems with few modifications to the runtime logic itself. In our initial effort, we focus on detecting interference patterns and informing applications about such cases, along with recommendations for better thread placement decisions. In the future, we also plan to explore the ability to give better predictions for the degree of parallelism an elastic runtime system should use in order to meet SLA requirements, without excessively overprovisioning resources. Elastic runtime systems refer to runtimes such as OpenMP or Cilk, which are able to quickly adapt to dynamic changes in the number of available cores.

In the next section, we will analyze how we can use performance counters to detect interference and how it can affect existing runtime systems and program execution times.

4. CASE STUDIES: INTERFERENCE

Modern machines often consist of several multicore processors. Resources commonly contended by processes include memory bandwidth as well as L3 and L2 cache. In this section, we present two case studies where performance counters can aid the identification of root causes of interference. In particular, we show processes which exhibit contention for memory controllers and L3 cache.

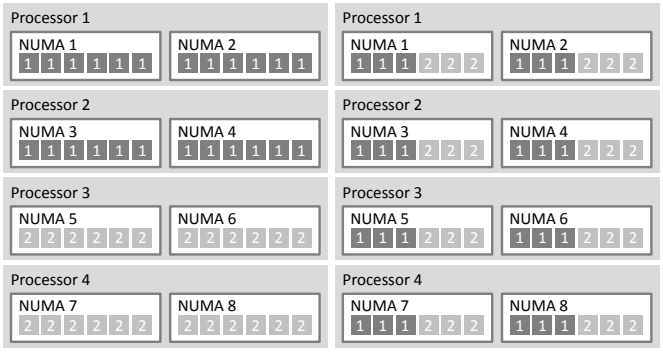


Figure 2: Thread assignment strategies used in Fig. 3 (left) and Fig. 4 (right).

Counters were read with `perf_event_open` in sampling mode at a frequency of 100 samples per second for both case studies. The samples were then aggregated in 1-second intervals.

4.1 Memory interference

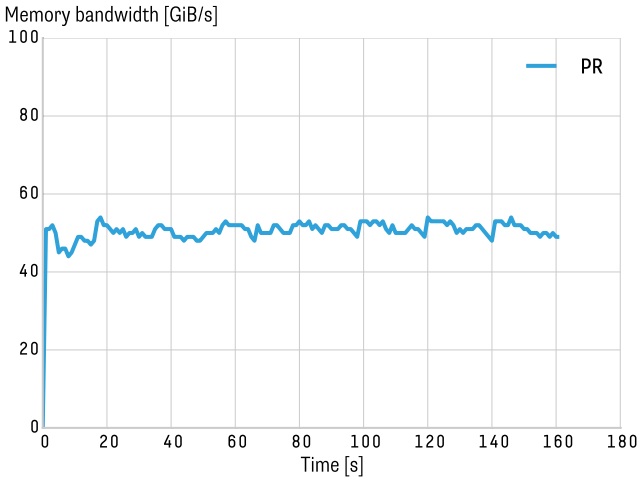
Memory interference is caused by applications with working sets unable to fit inside the CPU cache and/or poor locality of reference. Typically, such applications have higher L2 and L3 miss rates which result in an increased number of main memory accesses. This in turn can lead to contention on memory controllers and/or interconnect paths.

To show the effects of memory interference, we measured memory bandwidth of the PageRank (PR) [9] implementation as provided by Green-Marl [21] on a machine with four AMD Opteron 6174 processors and 128 GiB of RAM running Linux 4.1.12. Every processor comprises two 6-core dies, where each die is a separate NUMA node and has its own L3 cache and memory controller.

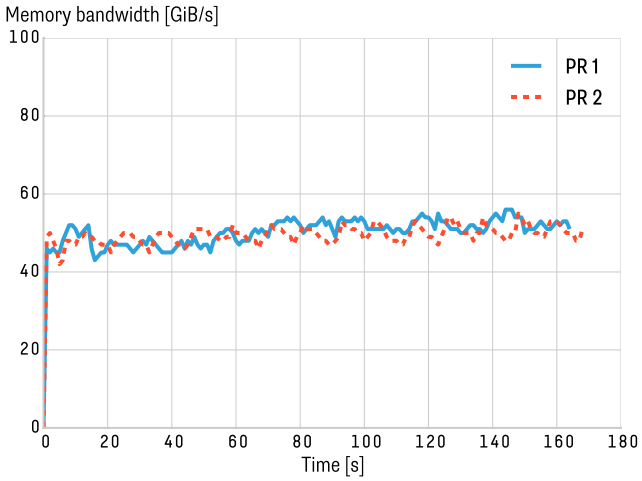
Memory bandwidth was calculated using a formula obtained from an AMD white paper [17]. The event "DRAM Accesses" was used to count memory accesses [1].

We ran two benchmarks: (1) one instance of PR running alone and (2) two instances of PR running simultaneously. Each instance was configured to spawn 24 threads and used a graph based on the Twitter social network [26]. The threads were pinned to cores during execution to allocate memory controllers and avoid interference by the Linux scheduler. We ran the experiment with two different thread allocation strategies (Figure 2). The first strategy assigned four entire NUMA nodes to each PR instance. For the second strategy, each PR instance was allocated three cores on every NUMA node.

In the first experiment, the execution time for a single PR instance running alone is around 162 seconds, and the measured memory bandwidth is approximately 51 GiB/s (Figure 3a). If two PR instances are executed together, the execu-

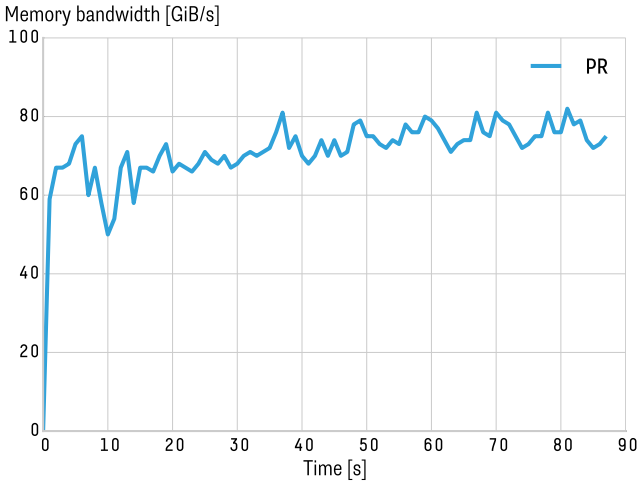


(a) A single PR instance running alone.

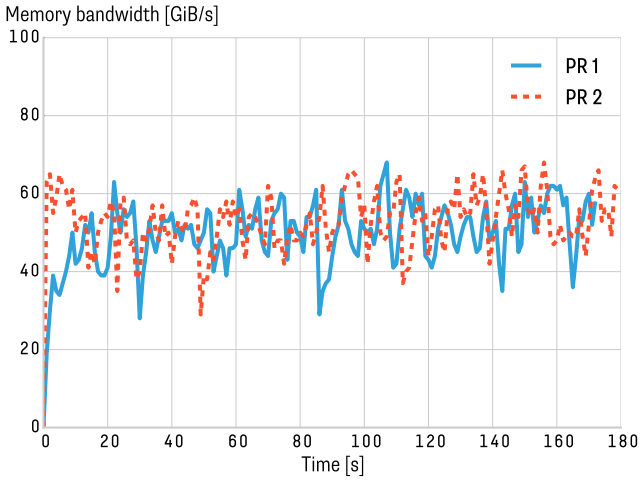


(b) Two PR instances running simultaneously.

Figure 3: Memory bandwidth of PR instances distributed on 4 NUMA nodes.



(a) A single PR instance running alone.



(b) Two PR instances running simultaneously.

Figure 4: Memory bandwidth of PR instances distributed on 8 NUMA nodes.

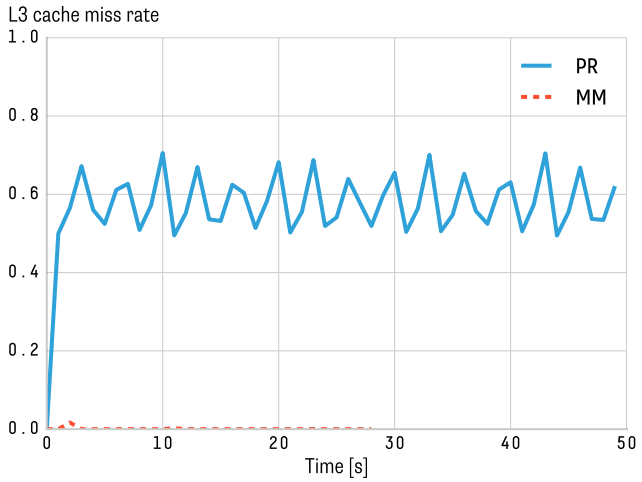
tion time and memory bandwidth remain roughly constant. The two instances do not interfere with each other since the threads of instances are assigned to separate sockets.

In the second experiment, each PR instance receives half of every NUMA node, i.e., each instance gets three cores on all eight dies. We observe the execution time of a single PR instance is nearly halved (Figure 4a). We attribute this to the fact that we now have the ability to make use of twice the number of memory controllers in the system. We see that the memory bandwidth is roughly 21 GiB/s higher than in Figure 3a. However, once we execute two PR instances simultaneously on the machine (with each instance now using 3 cores per NUMA node), we see that the execution time goes up to approximately 176 seconds and the memory bandwidth decreases to around 52 MiB/s. Assuming the two instances share memory controllers fairly, each instance receives the equivalent of $0.5 \times 8 = 4$ memory controllers. The execution time and memory bandwidth attained by instances therefore appears to be dependent on the number of available memory controllers.

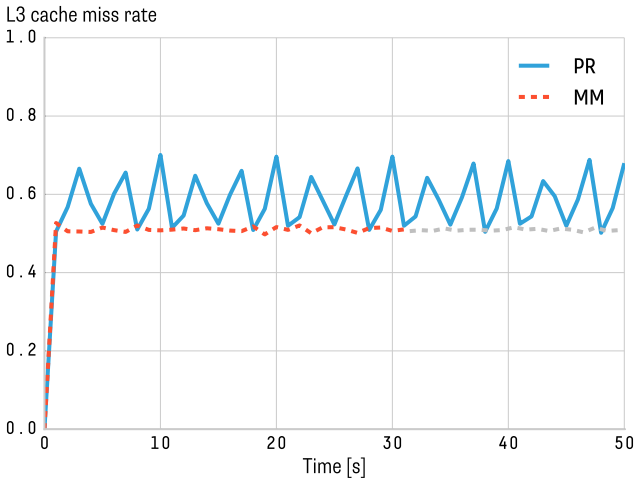
4.2 Last-level cache

In this case study we paired an application with good locality of reference and an application with poor locality to demonstrate contention on the last-level cache.

We ran matrix multiplication (MM) with PageRank (PR) on a machine with two Intel Xeon E5-2670 v2 processors and 256 GiB of RAM running Linux 4.2.0. For matrix multiplication, we used the ATLAS library which provides an implementation of BLAS for C [45] and the matrix MathWorks/tomography from the University of Florida Sparse Matrix Collection [15]. The matrix was multiplied by itself for 1000 iterations. The PR application is identical to the one in the previous case study and used the same Twitter graph [26], except it was configured to use eight threads. Each PR thread and the MM process were pinned to separate



(a) PR and MM instances running alone.



(b) PR and MM instances running simultaneously.

Figure 5: L3 cache miss rates of PR and MM instances.

cores on the same processor. The following two events were used to calculate the L3 cache miss rate [23]:

- `LONGEST_LAT_CACHE.REFERENCE`
- `LONGEST_LAT_CACHE.MISS`

The L3 cache miss rates of the applications running alone are shown in Figure 5a. MM and PR have average cache miss rates of 0.0008 and 0.57, respectively. The low miss rate for the former can be explained by the fact the input and output matrices are both approximately 2 MB each and fit into the processor's L3 cache (25 MB). In contrast, PR has a much higher cache miss rate which explains consumption of memory bandwidth in the previous case study.

The results of the applications running side by side are presented in Figure 5b. The cache miss rate of MM dramatically increases to 0.57 whereas the rate for PR remains steady at 0.58. However, the elevated cache miss rate resulted only in an 8.9% increase in execution time for MM. Because the overall execution time of MM is shorter than PR, we restarted MM after the program finished in Figure 5b. The continued execution is indicated by the gray line.

At a glance, the increases in the cache miss rate and execution time appear to be disproportionate; however, examination of the L2 cache miss rate clarifies this—it is relatively low at 0.07. As only a small number of requests reach L3 cache, an increase in L3 cache misses has limited effect on the execution time.

The following events were required to calculate the L2 cache miss rate [23]:

- `L2_RQSTS.ALL_DEMAND_DATA_RD`
- `L2_RQSTS.DEMAND_DATA_RD_HIT`
- `L2_RQSTS.ALL_RFO`
- `L2_RQSTS.RFO_MISS`
- `L2_RQSTS.ALL_CODE_RD`
- `L2_RQSTS.CODE_RD_MISS`

Unlike the L3 cache, there is no event which counts all the different types of L2 references and misses collectively, so they must be measured separately.

As mentioned before, Intel processors generally only have eight programmable performance counters per core. Consequently, the data for the L2 cache was collected in a separate run with identical configuration.

4.3 Discussion

The identification of interference is non-trivial. First of all, it requires domain knowledge about various CPU architectures. To illustrate, Intel and AMD provide different events as they are not standardized; events even vary between different generations of processors in the same family. In §4.2, we saw that big changes in one observed event had little impact on the overall program performance due to interactions with the rest of the system. Therefore, it is crucial to consider complex combinations of events for anomaly detection.

While most processors support a common set of frequently used events, they are not always identical. For instance, an event which counts the number of LLC misses may or may not include prefetches. Users must be aware of these details to obtain correct performance data. Furthermore, processors often lack events which are integral to performance analysis. For example, examination of the code of Intel Performance Counter Monitor [46] reveals it uses uncore events which monitor the integrated memory controller to measure memory bandwidth. The reliance on uncore events is a significant drawback since we cannot determine the memory bandwidth of a specific core or thread.

Even without the aforementioned issues, the selection of events to measure is a challenge in itself. This is further complicated because it does not scale if the process has to be repeated for every new processor model. However, interference anomalies do generate specific patterns which a learning algorithm can learn in order to do classification without detailed knowledge about specific events.

5. DESIGN

We now briefly describe the early design of Glatt and explain how we plan to solve the challenges listed in §3. On a high-level the design of Glatt comprises three parts: (a) interference detection, (b) interference classification, and (c) the recommendation of thread placements for runtimes.

A key design decision we take in Glatt is to assume no prior knowledge about the available hardware events on a given machine, except for how to configure and read counters, and which event codes are supported on the machine. While this appears restrictive, it greatly simplifies porting Glatt to different architectures or CPU models and removes the need to encode domain knowledge about event semantics in the Glatt code. The cost of this decision is that we need a training set for any given problem we want to detect (e.g., interference). However, we argue that this training set can be automatically evaluated using an appropriate set of benchmark programs, and need only be done once for any given machine. For example, in our initial test set we used five benchmark programs: PageRank, sort, matrix multiplication, hop distance, and single-source shortest path. Next, we ran these algorithms alone and pairwise on a machine with different thread allocation strategies as well as different working set sizes to try and force certain interference behaviors. We used repeated runs of the same settings to measure all available hardware events for our initial study and recorded the overall program performance. With the recorded data set we try to identify a set of events that is relevant for a given machine and benchmark. Initially, the event space can be reduced using statistical methods such as computing the correlation coefficients or principal component analysis to find a set of uncorrelated events.

Although a number of use cases exist for Glatt, we focus on detecting destructive performance interference between parallel runtimes, and providing corresponding feedback to applications. A first requirement for this is a useful measure of application performance itself. One option is to compare instructions per cycle (IPC) with previous runs of the same application and/or thread group. IPC seems to work well as a performance benchmark for certain workloads [49] but others find that it does not necessarily correlate with how well an application is doing [2]. A more reliable approach would be to modify the application or runtime to explicitly inform Glatt periodically about progress. To identify specific types of interference, our initial approach classifies the sample runs measured in our training set with the appropriate type. Online detection is then achieved by leveraging statistical classifier algorithms like support vector machines to identify the type of interference at runtime. The challenge is to generate a training set that provides accuracy across a wide range of different interference patterns such as contention for LLC, bandwidth, and SMT threads and to make the training set relatively independent of the architecture. Glatt will implement an upcall mechanism to inform runtime systems about potential interference and recommendations for thread placement changes.

For a complete system, we plan to record any data measured by default and make it available to the system through log files with additional metadata (e.g., per process information, binary names, etc.). Ideally, this knowledge can be be taken into consideration to improve the classification and decision algorithms of the given system in the future. Finally, typical programs have several phases, unlike benchmark programs in a training set. While several algorithms already exist to detect program execution phase changes, initially we will rely on the runtime systems' domain knowledge to inform Glatt about different phases and the relationship between threads (e.g., which threads form a parallel task). Automatically detecting groups of interacting threads is an interesting problem, but out of scope at this stage.

6. CONCLUSION

In this paper, we have shown how appropriately-chosen measurements of hardware events can be used to predict, and minimize complex interference between parallel runtimes on a multicore machine.

However, we have also highlighted a critical obstacle to using this technique: The increasing complexity of the monitoring facilities available on modern processors, combined with the tremendous variation across processor families and vendors, makes it almost impossible to portably exploit the available information at runtime.

In response, we are pursuing a design which learns an online model of the current machine, identifies measures which are useful predictors of interference, and uses these to recommend optimal thread allocations to user-space applications.

Our ultimate goal is to extend the use of hardware performance counters from specialized, skilled performance debugging to general purpose online scheduling.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Yungang Bao, for their helpful suggestions. We would also like to thank our university colleagues and our industry partners Hewlett Packard Enterprise, Huawei European Research Center, Cisco, VMware, and Oracle for their support.

8. REFERENCES

[1] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*. April 2010.

[2] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.

[4] Autumn. leaf: Open Machine Intelligence Framework for Hackers. https://github.com/autumnai/leaf, Apr. 2016.

[5] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 101–110, 2005.

[6] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Operating Systems Review*, 43(2):56–65, Apr. 2009.

[7] K. A. Bare, S. Kavulya, and P. Narasimhan. Hardware performance counter-based problem diagnosis for e-commerce systems. In *2010 IEEE Network Operations and Management Symposium*, pages 551–558, Apr. 2010.

[8] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 103–116, 2008.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, Apr. 1998.

[10] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, 2000.

[11] V. C. Cabezas and M. Püschel. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 222–231, 2014.

[12] Caffe. Caffe: Deep learning framework . http://caffe.berkeleyvision.org/, Apr. 2016.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, 2005.

[14] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 76:1–76:10, 2013.

[15] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, Dec. 2011.

[16] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 2003.

[17] P. J. Drongowski. Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf, September 2008.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[19] Google. TensorFlow: Open Source Software Library for Machine Intelligence. https://www.tensorflow.org/, Apr. 2016.

[20] N. Herath and A. Fogh. These Are Not Your Grand Daddys CPU Performance Counters. https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf, Aug. 2015.

[21] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, 2012.

[22] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, Apr. 2015.

[23] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. April 2016.

[24] A. Kleen. Intel PMU profiling tools. https://github.com/andikleen/pmu-tools, Apr. 2016.

[25] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, May 2008.

[26] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[27] D. Levinthal. PMU event analysis package. https://github.com/David-Levinthal/gooda, Apr. 2016.

[28] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, 2015.

[29] J. M. May. MPX: Software for multiplexing hardware performance counters in multithreaded programs. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, Apr. 2001.

[30] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 153–166, 2010.

[31] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, 2010.

[32] G. Ofenbeck, R. Steinmann, V. C. Cabezas, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 76–85, 2014.

[33] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, Nov. 2015.

[34] Oracle. Fork/Join. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, February 2016.

[35] T. Roehl. likwid: Performance monitoring and benchmarking suite. https://github.com/RRZE-HPC/likwid, Apr. 2016.

[36] J. C. Saez, J. Casas, A. Serrano, R. Rodríguez-Rodríguez, F. Castro, D. Chaver, and M. Prieto-Matias. *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, chapter An OS-Oriented Performance Monitoring Tool for Multicore Systems, pages 697–709. Springer International Publishing, 2015.

[37] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas. The impact of hyper-threading on processor resource utilization in production applications. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, 2011.

[38] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham,

T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[39] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News*, 37(2):46–55, July 2009.

[40] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 121–132, 2009.

[41] S. J. Tarsa. *Machine Learning for Machines: Data-Driven Performance Tuning at Runtime Using Sparse Coding*. PhD thesis, 2015.

[42] The Barrelfish Project. Barrelfish Operating System. www.barrelfish.org, Apr. 2016.

[43] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 75–84, 2009.

[44] V. Weaver. Linux Programmer's Manual: PERF_EVENT_OPEN(2). http://man7.org/linux/man-pages/man2/perf_event_open.2.html, May 2015.

[45] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[46] T. Willhalm, R. Dementiev, and P. Fay. Intel Performance Counter Monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor, Apr. 2016.

[47] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, Apr. 2009.

[48] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 23–32, April 2009.

[49] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, 2013.

[50] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, 2010.

[51] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):4:1–4:28, Dec. 2012.