# Shared Resource Sensitivity in Task-Based Runtime Systems

German Ceballos
Uppsala Universitet
german.ceballos@it.uu.se

David Black-Schaffer
Uppsala Universitet
david.black-schaffer@it.uu.se

## ABSTRACT

Task-based programming methodologies have become a popular alternative to explicit threading because they leave most of the complexity of scheduling and load balancing to the runtime system. Modern task schedulers use task execution information to build models which they can then use to predict future task performance and produce better schedules.

However, while shared resource sensitivity, such as the use of shared cache, is widely known to hurt performance, current schedulers do not address this in their scheduling.

This work applies low-overhead techniques for measuring resource sensitivity to task-based runtime systems to profile individual task behavior.

We present results for several benchmarks, both in an isolated environment (all resources available) and in normal contention scenarios, and establish a direct quantitative correlation between individual tasks and the entire application sensitivity.

We present insight into areas where these profiling techniques could enable significant gains in performance due to better scheduling, and conclude what scenarios are necessary for such improvements.

## Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes

## General Terms

Measurement, Performance

## Keywords

Resource Sharing, Tasks, Performance

## 1. INTRODUCTION

Task-based programming has become a compelling model for formulating and organizing parallel applications. Compared to thread-based programming, tasks provide several advantages, such as simpler load balancing and dependency handling, matching the exposed parallelism to the available resources, and low-overheads for individual task startup and tear-down. Compared to creating new threads, tasks typically start, schedule and clean up 10 to 20 times faster [6]. This reduced overhead allows for much finer-grain parallelism.

Popular implementations include Intel's Thread Building Blocks [6], OmpSs [8] and StarPU [1]. These frameworks are designed to exploit parallelism across the available resources by scheduling tasks to the available cores. This approach maximizes the CPU utilization per core, while avoiding problems with over or under-subscription.

Schedulers are crucial entities in these type of environments, as proper task placement and ordering directly impact execution time. A data dependency graph is usually automatically inferred from task descriptions, thereby allowing the system to identify tasks that can be run in parallel.

A great amount of work in scheduling has been done both for threaded and task-based systems. The most complex ones save information from previous executions into individual *performance models* and use them to improve scheduling decisions in the future [1]. However, none of these consider the impact of resource contention between the tasks themselves.

Recently, several techniques have been developed to measure performance information as function of available resources. The Cache Pirate [5], introduces a low overhead method for accurately measuring IPC and other metrics as a function of the available shared cache capacity. Collecting this information for individual tasks could reveal inter-task sensitivities to the scheduler, and thereby allow it to make more intelligent decisions at runtime.

This paper presents the results of using the Cache Pirate technique to profile per-task sensitivity to shared resource contention in the StarPU runtime framework.

By examining several task-based benchmarks we were able to determine that while some tasks are indeed sensitive to shared resource allocation, there is a limited opportunity to apply this knowledge due to the largely homogeneous nature of the tasks being executed at any given time.

## 2. EXPERIMENTAL SETUP

### 2.1 Methodology

The Cache Pirate technique is based on co-running an application that *steals* the desired resource from a *target* application. By carefully stealing just the desired resource, the effect of losing that resource on the target's performance can then be accurately measured via hardware performance counters. The recorded data includes miss ratio, miss rate, IPC, and execution time as a function of the available shared cache for the target application. This information can be used to predict application scaling. In task-based systems, however, special care must be taken to measure the data

*per-task*.

In [5] the pirate application was run on a separate core to steal shared cache from the target application. In a task-based runtime system, we have two choices to steal cache:

1. Executing the pirate as a task within the runtime system.

2. Co-running a separate pirate application along side the runtime system.

For (1), the pirate task should be submitted earlier than the other ones and executed during the whole execution of the task flow, or the runtime system should be modified to schedule a special pirate task separately from the regular tasks. An advantage of this method is its transparency with regards to starting or stopping the profiling stage. However, the runtime must ensure that the pirate task runs continuously and it is not possible to separate the shared resource overhead of the runtime itself from the measurement.

In (2), a pirate application is co-run with the runtime system, pinned to one core and affecting the whole system performance. This strategy fixes the resources for the runtime to one core less than the physical system, but does not require modifying the runtime.

In terms of recording profiling information we also had two approaches: either have each task recording its information when finishing execution, or synchronize the tasks with the external pirate application to store the data.

We used a mixed approach, co-running an external pirate application, while recording data for each task from within the runtime. We consider this alternative simpler when managing a large number of fine-grain tasks for two reasons. First, the overhead of running the runtime system will be the same for each task as we will be also be stealing cache to the runtime itself. Second, if we wrap every task with output routines, we are able to control the granularity and gather data for the exact amount of work in the task.

To do this we logged the task start and finish times and compared across several executions of different benchmarks.

## 2.2 Available Platforms

In order to retrieve fine-grained information, we needed the flexibility to modify the runtime system for data collection. We looked at both Intel's TBB and StarPU for this purpose.

Intel's Thread Building Blocks (TBB) is a powerful and portable library of parallel acceleration structures. The task declaration process is straightforward and the scheduler is reasonably simple. TBB's scheduler is clean and modularized, but the scheduling policy is mostly fixed, making it difficult to insert a special pirate task.

StarPU is designed with the goal of making it easy to experiment with different schedulers and policies. Some of the default policies are guided with *performance models*, that consist of information from past executions, used to extend the data dependency graph with weights. This approach is important to support StarPU's ability to automatically schedule tasks to heterogeneous devices (e.g., CPUs and GPUs) provided the developer defines appropriate tasks for each device.

Both frameworks provide micro-benchmarking suites with different problem set sizes. Parts of the PARSEC [3] benchmark suite have been ported to TBB.

The results presented in this work are based on StarPU, as its scheduler design provided the flexibility needed to collect shared resource sensitivity data.

## 2.3 Evaluation Framework

Data was collected on a quad core Intel Core i5-3550 CPU (*Ivy Bridge*) with a 32kB 8-way associative L1 cache, a 256kB 8-way associative L2 cache and a 6MB 12-way associative L3 cache.

The memory controller has two channels. The baseline setup uses four dual-ranked 4GB DDR3-1333 DIMMs, for a total of 16GB. The StarPU release used was version 1.05, together with Intel's TBB (version 4.2). To gather information from hardware performance counters we used the PAPI [7] library. The test applications were the ones provided in the default StarPU microbenchmarking suite, and the runtime system was configured with the *eager* scheduling policy.

The sensitivity of each task to its shared cache allocation was first measured by running the application pinned to a single core and running the pirate application on another core. This allowed us to understand how each task reacts to losing space in the shared cache. We also measured the performance of each task when the runtime was allowed to use all cores, giving us the actual execution in the presence of cache sharing from the other executing tasks. Performance counter measurements for the parallel executions were done for tasks pinned to a particular core to ensure that we did not measure events from other cores.

Data dependencies, interleavings and execution traces were available through StarPU's own profiling tools. Combining this data with resource sensitivity contributed to a better understanding of the executions.

# 3. EVALUATION

## 3.1 Single Task Execution

Figure 1 shows the relative degradation of performance (%IPC) for every non-trivial benchmark of StarPU. The x-axis shows the amount of shared cache available to the target application and the y-axis shows the average application IPC. Although we are ultimately interested in per-task sensitivity, this information helps us to identify the most cache-sensitive applications: `heat`, `cg`, and `block`. These benchmarks demonstrate a considerable slowdown as they lose shared cache. Meanwhile, others such as `ci`, `axpy` and `dot_product` are not sensitive at all, which is expected from

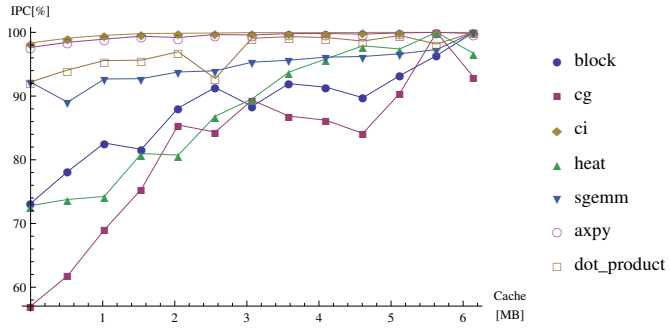


**Figure 1: Whole application sensitivity as a function of shared cache allocation.**

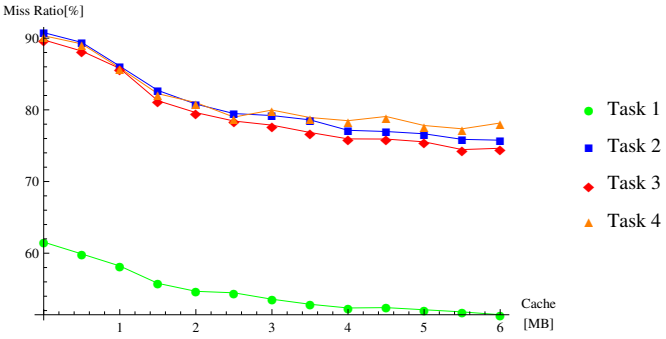


**Figure 2: Per-Task Miss Ratio Curves for the `heat` benchmark.**

computational-intensive applications with low memory footprint.

In general the curves show monotonic decreases in IPC but we do see some anomalies. The pirating technique works by stealing a certain number of ways of associativity of the cache. Isolated anomalies and negative trends (`ci` at 2.5MB) are caused mainly by the working data set, and the way it fits in cache. Some benchmarks split the original data set into tasks that suffer from reduced associativity more than others.

Other anomalies (such as `cg` at 6MB) are due to the pirate's inability to reliably steal that much cache. This means that we are not able to trust this data anymore, as the miss ratio of the pirate is high, indicating it is not stealing the cache effectively.

Focusing then on the interesting cases, we used a low overhead task wrapper that starts, stops and reads the performance counters for each task executed. The applications mentioned above contain up to five different tasks each, with medium or heavy degree of dependencies between them, causing a pyramidal dependency graph.

Figure 2 shows Miss Ratio Curve (MRC) for each task of the `heat` application, exposing their different knees and sensitivity levels. We can see that all tasks present a decrease of the miss ratio when having more cache available up until a certain point, from where the variance is minimum.

In almost every benchmark, the most frequent task (and usually the most time-consuming) is also the most affected by shared cache. This evidence leads us to conclude that a different task evaluation order or pinning could reduce resource contention and speedup the execution. Unfortunately, data parallel applications present extremely regular workloads with a large number of instances of the same tasks, resulting in a low *task diversity*.

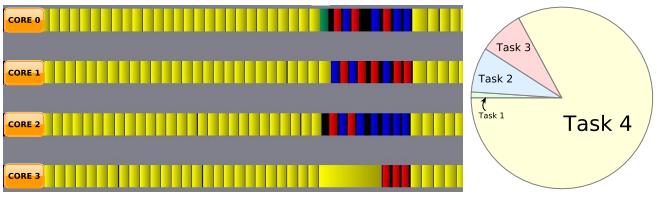


**Figure 3: Heat execution trace (left) and overall task diversity (right).**

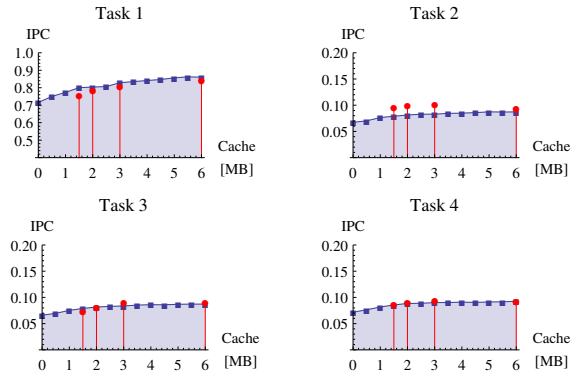


**Figure 4: Heat IPC Predicted vs Measured**

Figure 3 depicts this fact showing a fragment of an execution trace for the heat benchmark running on multiple cores (one core per line). Each color block shows the execution time of a task, with each task given a different color. It also shows proportion of instances for each task. As can be seen from this trace, for most of the execution time the application is dominated by one type of task. This low task diversity eliminates the possibilities of improving scheduling by taking resource contention into account as there are no scheduling options. It is only for the relatively short period of time where the schedule exhibits task diversity that resource contention information could improve the scheduling decisions.

## 3.2 Co-running Multiple Tasks

After gathering individual profiles for every task by running them on a single core with the pirate, the benchmarks were run again under different core configurations, exposing the effects of resource contention. By varying the number of cores used, we can effectively vary the degree of resource sharing. E.g., when run on four cores, the homogeneous tasks shown in Figure 3 would each receive $\frac{1}{4}$ of the 6MB shared cache, while when run on three cores they would each receive $\frac{1}{3}$. Figures 4 and 5 present the average IPC and execution time for each task reported when run on a single core with the pirate stealing the relevant amount of cache, compared to the actual IPCs measured when the application is executed in parallel.

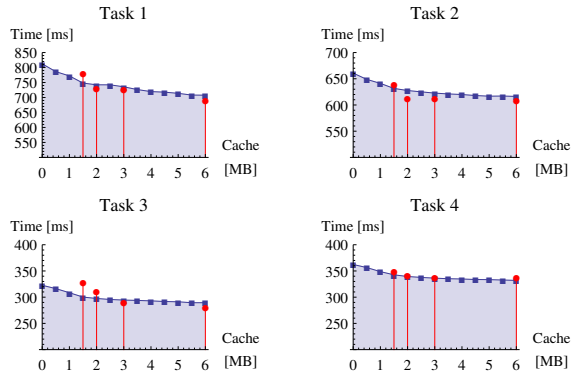


**Figure 5: Heat Execution Time Predicted vs Measured**

The results gathered from the pirate (*pirate's prediction*) is shown in blue, while the measured contention results are displayed in red. For 6MB of cache the application was run on a single core, for 3MB on two, etc. The data was gathered for an original problem size of 256MB.

Both figures demonstrate the accuracy of the pirate in task environments. The difference between the predicted and the measured range from 0.5% to 14% in worst cases, with an average error of 5%. For `cg` and `block` the slowdown is similar to the values illustrated.

Some of the errors are due to contention on some other resources such as bandwidth. As shown with per-task MRCs, an increase in the miss rate involves an increase in bandwidth consumption, thereby hurting the IPC.

At the same time, despite the cache pirate working accurately in single threaded benchmarks, it shows instability with cache configurations lower than 2MB. As we can see from the figures mentioned, results for 2MB or below are the ones affecting the technique's accuracy in this model.

For example in Figure 4, the actual IPC of Task 2 is better than the predicted by the pirate. It also presents a mostly flat segment in the MRC after 2MB. As Task 2 is being co-run with similar tasks, bandwidth consumption is increased, leveraging the reduced available cache and higher miss ratio facts, which ends up in higher IPC.

# 4. RELATED WORK

Yoo [9] presented work on resource-contention aware scheduling techniques, exploiting data locality at scheduling time, and techniques for modeling locality patterns. Bhadauria et. al. [2] and Blagodurov et. al. [4] present a very thorough analysis of co-scheduling impacts. Both of them are based on threaded workloads. Some qualitative work on improving task scheduling has been done in [1] that were later used to design the StarPU scheduler.

# 5. CONCLUSION

In this work we profiled and analyzed multiple task-based workloads for shared resource sensitivity in order to gain insights towards improving scheduling decisions.

The total application's sensitivity is correlated to the shared resource contention of its individual tasks, although in most cases only a few of them are highly sensitive. This fact could be leveraged by merging this type of quantitative data into the scheduling policies to avoid co-scheduling conflicting tasks.

Two main problems are identified in using shared resource sensitivity to improve scheduling. First, a significant degree of task diversity is required to be able to adjust the schedule to avoid shared resource contention. However, the provided benchmark applications showed little diversity for most of their execution.

Second, the actual slowdowns measured due to resource contention for co-running several tasks is lower than the benefit from adding cores to the runtime system. This means that even though per-task performance drops as more tasks share the cache, the performance increase from more parallel execution is still positive. In spite of this, potential bottlenecks could be generated when working with bigger working set sizes or different core-cache relations than our evaluation framework.

### *Can we do better?*

To cope with the task diversity problems, decisions could be made at the runtime level for *multiple* applications submitting tasks to the same runtime. This would increase the diversity of tasks available to the scheduler.

Systems with similar core configurations but smaller cache sizes or different cache hierarchies, or even different replacement policies, could exhibit worse results and lower performance under the same sensitivity profiles.

We believe that there is only a limited opportunity to apply this knowledge isolated due to the largely homogeneous nature of the tasks being executed at any given time. However, combining this approach with simultaneous measurements of bandwidth sensitivity, and establishing a more clear correlation of these factors with performance degradation, scheduling decisions could be improved specially when co-running applications to address the poor task diversity issue.

# 6. ACKNOWLEDGMENTS

This work was supported by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

# 7. REFERENCES

[1] C. Augonnet, S. Thibault, Namyst R., and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Concurrency and Computation: Euro-Par 2009*, pages 187–198, 2009.

[2] M. Bhadauria and S. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proc. ACM Intl. Conf. on Supercomputing*, pages 189–199, 2010.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, 2011.

[4] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, pages 8:1–8:45, 2010.

[5] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Intl. Conf. on Parallel Processing*, pages 165–175, 2011.

[6] Intel. *Intel Threading Building Blocks*, 2010. Doc. No. 319872-006US.

[7] S. Moore and J. Ralph. User-defined events for hardware performance monitoring. *Proc. of the Intl. Conf. on Computational Science*, pages 2096–2104, 2011.

[8] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. ACM Intl. Conf. on Supercomputing*, pages 142–151, 2008.

[9] M. Yoo. *Locality-Aware Task Management on Many-Core Processors*. PhD thesis, Stanford University, 2012.