Characterizing Task Scheduling Performance Based on Data Reuse

Germán Ceballos[†], Thomas Grass[‡], David Black-Schaffer[†] and Andra Hugo[†]
† Dept. of Information Technology, Uppsala University, Sweden
‡ Barcelona Supercomputing Center, Spain
german.ceballos@it.uu.se, thomas.grass@bsc.es, david.black-schaffer@it.uu.se, andra.hugo@it.uu.se

ABSTRACT

Through the past years, several scheduling heuristics were introduced to improve the performance of task-based applications, with schedulers increasingly becoming aware of memory-related bottlenecks such as data locality and cache sharing. However, there are not many useful tools that provide insights to developers about why and where different schedulers do better scheduling, and how this is related to the applications' performance. In this work we present a technique to characterize different task schedulers based on the analysis of data reuse, providing high-level, quantitative information that can be directly correlated with tasks performance variation. This flexible insight is key for optimization in many contexts, including data locality, throughput, memory footprint or even energy efficiency.

Keywords

Task-based Scheduling; Data Reuse; Cache-sharing

1. INTRODUCTION

With the growing complexity of computer architectures, scheduling task-based applications have become significantly more difficult. Typical approaches for optimizing scheduling algorithms consist of either providing an interactive visualization [4, 1] of the execution trace or simulating the tasks execution [10, 3] to evaluate the overall scheduling policy in a controlled environment. The developer has to analyze the resulting profiling information and deduce if the scheduler behaves as expected, and *qualitatively* compare different schedulers.

This is particularly difficult because the complex bad scheduling decisions are often seen as an effect on the performance difference between tasks of the same type. Existent work [11] proposed scheduling strategies that include these performance differences in the load-balancing algorithm to overcome the precision loss of the decision process. However, understanding the underlying causes of performance anomalies of the tasks as well as the snowball effect of the dynamic scheduler is still an open question.

In this paper, we present a new methodology to characterize, in a *quantifiable* way, the scheduling process in the context of one of the most important performance-related characteristics: how the schedule affects data reuse between tasks. We show how the data reuse pattern through the execution can provide insight to the performance of the scheduler, independent of what is optimizing for (locality, bandwidth, memory footprint, etc.).

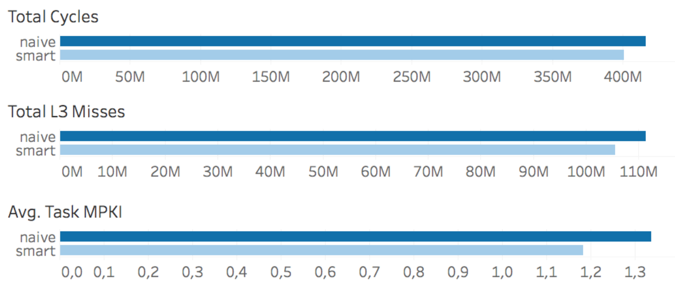


Figure 1: Performance difference between `smart-bfs` and `naive-bfs`.

Task-based schedulers look at the set of tasks that are ready for execution and choose which tasks should execute on which processor to minimize execution time. In order to understand the performance of a particular schedule, and thereby the scheduler itself, it is necessary to address two critical questions: (1) **What** were the scheduling decisions that influenced the performance of the execution, and (2) **When** did those decisions happen?. As the resulting performance of an application is mainly driven by the memory-bound phases, we correlate these questions to the reuses of memory accesses, building an inter-task data reuse graph. This allows us to quantify the scheduler's behavior by comparing the **actual** reused data against the **potential** reuses. We also observe the performance of the tasks over time, exposing quantitatively why tasks of the same type perform differently depending on the scheduler under a specific memory configuration (cache size).

2. MOTIVATION

We take as an example a task-based implementation of the Cholesky Factorization[1] within the OmpSs runtime [5], and we study its performance through time using a single threaded execution in the TaskSim simulator[2] [9, 8].

Fig. 1 [3] shows the total cycle count, total number of last level cache misses and average task misses-per-kilo-instruction (`mpki`), for two different simulated executions.

The only change between the executions is the scheduling policy of the runtime system: `naive-bfs` with a regular

[1]The input is a 32MB matrix with 256x256 block size. The application generates a total of 120 tasks of four different types (`gemm`, `potrf`, `syrk`, and `trsm`).
[2]Default configuration, using a 2MB last level shared cache.
[3]The reader can click on the figures to see an online interactive version.

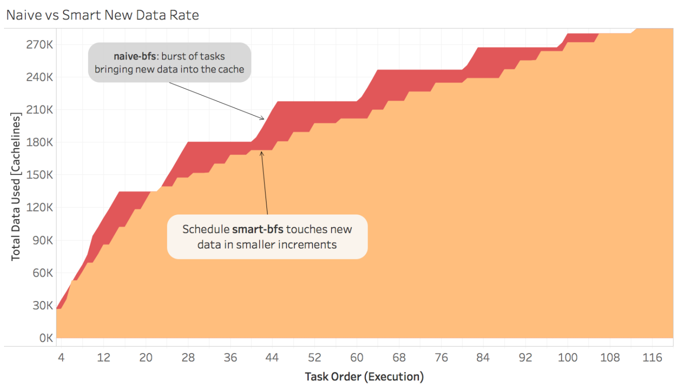


Figure 2: Differences in new data rate.

breadth-first-search policy, scheduling tasks in creation order, and `smart-bfs` which schedules tasks according to a heuristic breadth first search, where if the task currently running creates a child task, that path is prioritized compared to the next task in the bread-first order. This optimization metric is locality, through the assumption that child tasks will reuse the data from their parent.

As we see from the total cycle count, `smart-bfs` is 6% faster than the `naive-bfs` scheduler. From the cycle count breakdown we find out that the difference comes from the amount of cycles spent on DRAM accesses, as a result of a 5% increase in last-level cache misses. When looking at the average task mpki, we see that each task misses 14% more in the later case.

At this point, overall statistics are not enough to give us valuable insight on the reason **why** one of the schedules can incur more cache misses, **when** in the execution this happens nor **what** is the potential for improvement. Measuring hardware-specific statistics using performance counters provide overall values but cannot give any insight on the specific sequence of events that caused those effects.

Furthermore, there are no generic, framework-independent and architecture-agnostic tools or techniques available to analyze these issues in a straightforward way.

We propose a new technique to infer quantitative metrics that enable performance characterization of different schedules through the analysis of inter-task data reuses. First, we describe our technique that is only dependent on the schedule as an input parameter, which enables the insights to be architecture-independent. Then, we show how by adding a second parameter, the hardware architecture, the tool allows correlating changes in reuse patterns with memory behavior and its real impact on performance.

3. THROUGH THE DATA-REUSE GLASS

Each of the tasks in an application operates in a certain data set. Throughout the execution, part of this data might be reused by later tasks. This means that a portion of the data set can be considered *shared*. It is well known that it is important to reuse shared data soon in the execution, as it maximizes the chance to serve those memory accesses from the caches, taking advantage of the temporal locality of the data. The same holds for a task-based application, with the difference that a change in task schedule affects how *soon* the shared data will be reused.

To understand this impact, it is first necessary to analyze how much *shared* and *private* data each task has. We do this by profiling the execution to sample the memory addresses for each task. Once the data is collected, we propose the following classification. For a given schedule, every memory access for a task is either new (first time seen) or reused from a previous task.

With this observation, we can divide memory accesses into the following four categories:

- **New-data:** the first time the memory address is used in the application.
- **Last-reuse:** the memory address was also touched by the task previously executed.
- **2nd-last-reuse:** the memory address was touched by the second-to-last task.
- **Older-reuse:** the memory address was used before by an older task.

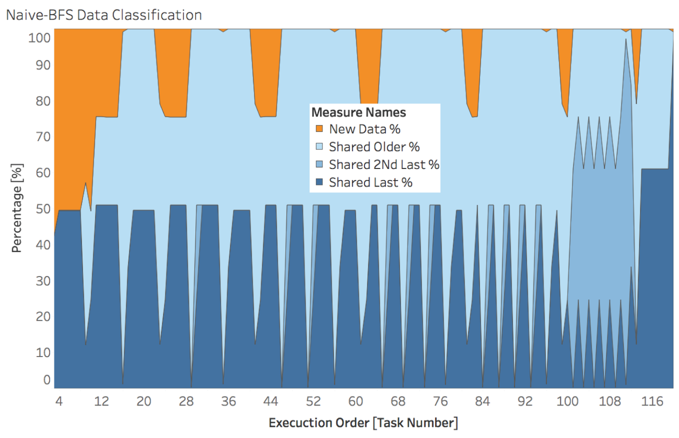
Figure 2 compares the cumulative amount of data touched as the program executes between the two schedules. Note that the total number of accesses in the `new-data` category is a function of how much data the application uses, and as a result both schedules bring in the same total by the end of execution. In the figure we can see how the `naive-bfs` schedule (red curve) executes tasks in a way that touches new data much more aggressively, in bursts. On the other hand, the `smart-bfs` schedule (orange curve) is much smoother, meaning that new data is brought at a much slower rate. The flat regions on the curves indicate *reuse-periods*, where the scheduled tasks operate on previously used data, and therefore does not bring in any `new-data`. From this data it is clear that the different schedules result in tasks reusing data in significantly different patterns, which will result in different cache miss rates, and therefore impact performance.

Although the `new-data` category intuitively exposes the *rate* at which the applications install new data in the caches, it is not explaining how the shared data is being used. Thus, to understand the details of how the two schedules reuse data differently, we need to look at the other memory access categories.

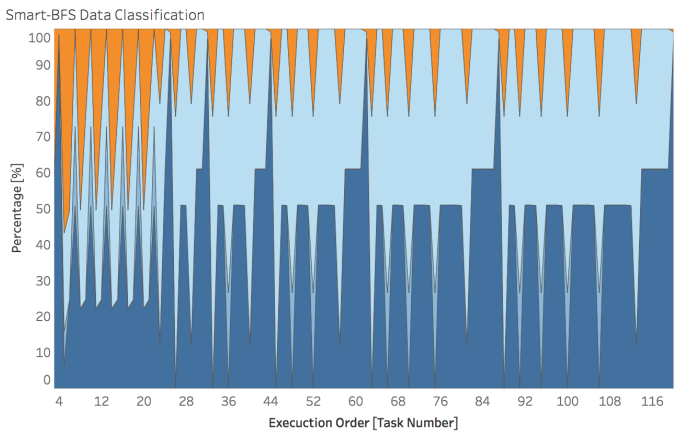
Figures 3a and 3b show the breakdown of memory accesses (`new-data`, `last-reuse`, `2nd-last-reuse`, `older-reuse`) for both schedules (`naive-bfs`, `smart-bfs`) as a function of time (task scheduled). The `last-reuses` are shown on the bottom (dark blue), while the upper area (orange) represents the new data regions. The middle-bottom region shows the percentage of second-last memory accesses, and finally the middle-top region (light blue) displays the relative amount of data reuses that come from older tasks, `older-reuse`.

The first thing we notice is that the area corresponding to `new-data` is distributed more sparsely across the graph for the `smart-bfs` policy, compared to the `naive-bfs` approach, where most `new-data` is touched during the first 16 tasks. In addition, the area corresponding to last-reuses increased considerably (more dark blue area), meaning that more shared data is being reused sooner in `smart-bfs`. This is also observed between the period of tasks 100 to 115: in the `naive-bfs` schedule, most of the data used was coming from the second-last predecessor, but in the smart schedule, now data is coming from its immediate predecessor.

With this classification, it is now clear that the two schedules have both very different reuse characteristics. However, we need to translate the observations from the previous figures into relevant metrics to compare the schedules in a



(a) naive-bfs schedule



(b) smart-bfs schedule

Figure 3: Relative data reuse.

straightforward way. We do this by looking at aggregated statistics from each reuse category over time, and understanding how reuses flow from one category to another.

Figure 4 shows an example of this. It displays for each task executed the percentage of memory accesses corresponding to each category (y-axis, %), as a function of time (x-axis; task number, i.e. time), for both `naive-bfs` (left) `smart-bfs` (right) schedules. The average value (%) for each access category is displayed with the *Average* line. By comparing the averages, it is possible to see that the `smart-bfs` scheduler has 11% more `last-reuses` reuses than `naive-bfs`. Most importantly, this view of the execution allows us to understand the effect of these changes: we can see that 5% of the execution time increase comes from the `smart-bfs` schedule turning second-last reuses into last-reuses, while the remaining 6% comes from improving older reuses.

Furthermore, this graph not only allows us to quantitatively see *why* the schedules are different, but, more importantly, it also shows us *when* their differences cause changes in data reuse. Since the tasks are uniquely identified, this analysis allows us to point to the specific tasks that benefited from the rescheduling or were hurt by it.

The insights from this analysis now allow us to understand the impact of scheduling changes in a way that can be used by schedulers to improve performance by increasing reuse through caches. Approaches that only measured the actual cache miss ratios per task (e.g., using hardware performance counters) are unable to trace back changes in memory behavior to the scheduling decision that caused them in this manner. As a result, this is the first analysis that allows scheduler designers to gain insight into how specific scheduling decisions impact later tasks.

4. EVALUATION AND ANALYSIS

We implemented a tool using Intel's Pin [6]. During execution, an *address map* is created where for each memory address accessed (cacheline granularity), a list of the corresponding tasks using it is kept, capturing execution order (schedule). The overhead of the Pin tool less than 20%.

Later, an analysis phase is run on the collected data. A *data reuse graph* is built, which is a unique representation of the applications' data characteristics, not dependent on the schedule. Each node represents a task instance, while each edge represents the amount of data shared between those tasks. In addition, a list of the unique memory addresses (datasets) is kept for each task instance.

By having the schedule as an input, it is possible to walk though this graph, analyzing each task dataset and comparing it to the previous tasks. The graph is very dense, but it is only necessary to walk through it according to the input schedule. What is more, since the representation is schedule agnostic, it is possible to walk the same graph in a different fashion, using a different input schedule and concluding the correct results. This enables the prediction of the sharing under different schedules without the need for re-profiling the application.

This technique allows characterizing the impact of different schedules in memory behavior in a high-level, framework- and hardware-agnostic way. However, comparing the relative differences between these metrics (per-category) is not enough to conclude what the effects are going to be in performance. Thus, we describe how by combining our reuse model with the hardware architecture properties we are able to provide insight for optimization for a specific platform.

In this study, we consider optimizing for cache locality and we measure the performance of each task by considering the cycles-per-instruction (CPI). In Fig. 4 we observe the CPI for every scheduled task, color coded by its type. We can see that in the naive schedule, the variation of CPI across the tasks is much more than in the smart case (average 20% variation).

For instance, if we look at Task 57, in the naive approach it is scheduled in the 57th position, resulting in 0.33 CPI. On the other hand, in the smart schedule, this task was executed 32nd, and resulted in 0.29 CPI, meaning a 15% difference in performance.

As our technique can tell exactly where the data is coming from, by correlating CPI with our reuse analysis we are able to see (highlighted in the figure) that in the first schedule, the task is reusing 98% of its data from older tasks. By scheduling this task much sooner, like in the smart schedule, we see that now 96% of the data is coming from the previously executed task, increasing the likelihood that the data is reused through the cache, instead of bringing it from main memory.

Moreover, we also see different performance within the same schedule for tasks of the same type: e.g. in the smart schedule, tasks 113-118 are all `syrk` with same size of input, however, task 113 has a 7% worse CPI than its subsequent ones, and by correlating with the *new data* graph, we see

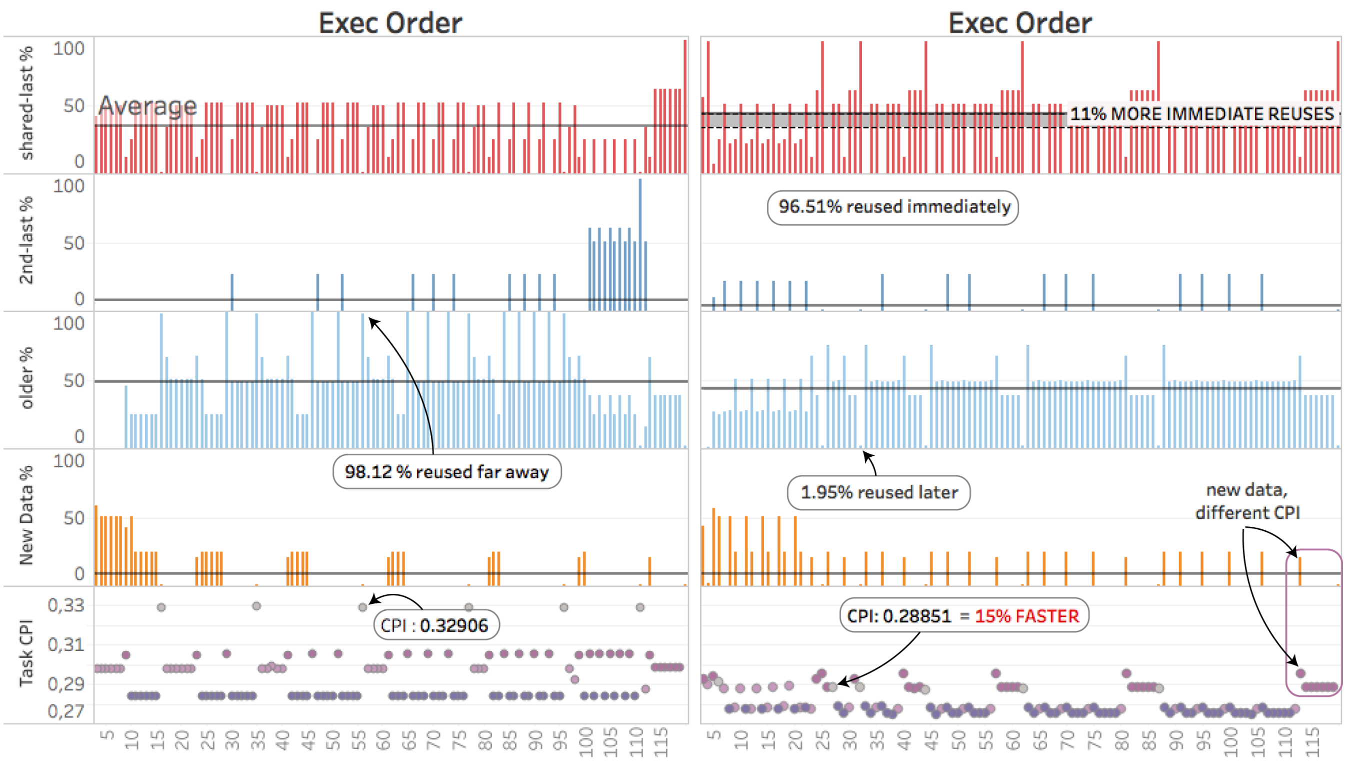


Figure 4: Breakdown of reuses per category: naive (left) vs smart (right).

that this is because it is bringing 20% new data.

By knowing the size of the last level cache, and by looking at the amount of new data per task, it is possible to estimate how many tasks can be scheduled, and which of them, before the data that is going to be reused is evicted. When *optimizing for locality*, this kind of insight is critical: if data is never going to be touched again, it is possible to schedule a task that brings new data into the cache; if the data is not going to fit in the cache anyways, tasks can be scheduled later in order to prioritize those that reuse data already present. This information can also be used to determine the *task granularity* as the memory reuse between tasks can influence directly their size.

Another example is *optimizing for memory throughput* on GPUs. The naive schedule results in lower performance on CPU, but it has a good property for the GPU: it clusters many tasks that touch new data. This can be useful for scenarios where data transfer is expensive and it is better to amortize the transfer cost by moving more data at once, as in GPUs. Our analysis gives this insight in a high level way, while being transparent to the targeted architecture.

5. RELATED WORK

Previous work has proposed different ways to diagnose scheduling anomalies by either interactively visualize information ([4, 1, 7]) or by simulating the task execution in order to provide a deterministic behavior of the scheduler ([10, 3]) without evaluating the performance behavior as a result of the memory use. On the other side other significant effort has been done to study the locality as a metric to characterize the workload of an application ([12, 2]) without considering the ad-hoc scheduling decisions taken as a result of the complex architectures. In our work we characterize the scheduling behavior as a result of the memory reuse. We provide quantifiable insight on how two schedulers behave differently and on how scheduling decisions of a task-based application affect the performance of task instances.

Drebes et al. [4], as well as other different visualization tools ([1], [7]) propose summarizing and averaging information provided by both the runtime and the hardware performance counters. By integrating this data in an interactive visualization tool, the programmer can observe the order of execution of the tasks, their duration, data dependencies, status of the computing resources, etc. However, when certain tasks end up executing in a certain order and with different performance, it is up to the programmer to reverse engineer the scheduler's decision, the reasons behind them and the points where those decisions happened, triggering the current behavior. Our work proposes a solution to help the programmer understand the variation in performance across the tasks, based on the analysis of memory reuse, capable of showing the exact points in time and underlying reasons for this variation.

Stanisic et al. [10] as well as Chronaki et al [3] rely on simulation of the tasks' execution in order to isolate the scheduler's effect on performance from tasks' unpredictable behavior. Our work also works with native executions of the entire application, characterizing the interaction between the scheduler and the tasks. Thus we are able to understand how tasks affect each others performance due to memory reuse and how the dynamic scheduling decisions are affected.

Tillenius et al. [11] observed that tasks of the same type have different executions time and estimated task sensitivity to resource sharing. Based on this they adjusted the scheduling in order to optimize the execution time of the application. In our work we analyze the reason for these performance difference, i.e. the way the tasks are reusing the memory, and we provide quantifiable information that can be used by the scheduler in order to avoid such effects and improve the precision of the dynamic decisions.

Weinberg et al. [12], as well as Cheveresan et al. [2] propose to use *memory reuse* as a metric to characterize workloads. Through this technique they analyze spatial and temporal locality of the application independent of the architecture. Our work uses a similar approach but with a more concrete purpose, using this information to understand the scheduling of task based applications, and implicitly correlate it to performance, memory aware optimizations (or energy in future work) depending on the metric of the scheduler.

6. CONCLUSION

In this work we propose a methodology to provide the programmer with high-level, quantifiable information regarding the scheduling decisions of a task-based application. We use a classification of the memory reuses through time to diagnose a scheduling decision, understanding the effects it triggered (what) and at which point in time this happened (when). This insight is critical as it can determine the reason (why) for the performance variation of tasks of the same type and for scheduling decisions based on it.

In this paper, we use this technique to characterize two different scheduling policies of a task-based implementation of the Cholesky Factorization, showing that they are reusing data in different ways when optimizing for data locality. Depending on the architecture on which they run (the cache size) we observe performance differences that correlate directly with the memory reuse information.

By collecting quantifiable information of the schedule's behavior we open up to future work on exposing this insight to different scheduling policies. We target using this information to optimize for locality (for NUMA aware architectures), bandwidth (for CPU/GPU architectures), memory footprint or even energy efficiency. Further extensions will consist in extending this tool to multi-threaded applications, exploiting its ability to predict information of other schedules with low overhead.

7. ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work was supported by the Swedish Foundation for Strategic Research project FFL12-0051, the Swedish Research Council Linnaeus UPMARC centre of excellence. Furthermore, this paper was published with the support of the HiPEAC network that received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 687698. Special Thanks to Trevor Carlson, Marc Casas and Miquel Moretó Planas.

8. REFERENCES

[1] R. Bell, A. D. Malony, and S. Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2003.

[2] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of workloads used in high performance and technical computing. In B. J. Smith, editor, *Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007*, pages 73–82. ACM, 2007.

[3] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In L. N. Bhuyan, F. Chong, and V. Sarkar, editors, *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 329–338. ACM, 2015.

[4] A. Drebes, A. Pop, K. Heydemann, and A. Cohen. Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, pages 274–283. IEEE Computer Society, 2016.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[6] 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, 2005.

[7] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.

[8] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramírez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *TACO*, 8(4):36, 2012.

[9] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramírez, and M. Valero. Trace-driven simulation of multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA*, pages 87–96. IEEE Computer Society, 2011.

[10] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J. Méhaut. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015.

[11] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-aware task scheduling. *ACM Trans. Embed. Comput. Syst.*, 14(1):5:1–5:25, Jan. 2015.

[12] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 50–, Washington, DC, USA, 2005. IEEE Computer Society.