Modeling the Interactions Between Tasks and the Memory System

Germán Ceballos
Modeling the Interactions Between Tasks and the Memory System

Germán Ceballos
german.ceballos@it.uu.se

October 2017

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Computer Science

© Germán Ceballos 2017
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden
Abstract

Making computer systems more energy efficient while obtaining the maximum performance possible is key for future developments in engineering, medicine, entertainment, etc. However, it has become a difficult task due to the increasing complexity of hardware and software, and their interactions. For example, developers have to deal with deep, multi-level cache hierarchies on modern CPUs, and keep busy thousands of cores in GPUs, which makes the programming process more difficult.

To simplify this task, new abstractions and programming models are becoming popular. Their goal is to make applications more scalable and efficient, while still providing the flexibility and portability of old, widely adopted models. One example of this is task-based programming, where simple independent tasks (functions) are delegated to a runtime system which orchestrates their execution. This approach has been successful because the runtime can automatically distribute work across hardware cores and has the potential to minimize data movement and placement (e.g., being aware of the cache hierarchy).

To build better runtime systems, it is crucial to understand bottlenecks in the performance of current and future multicore systems. In this thesis, we provide fast, accurate and mathematically-sound models and techniques to understand the execution of task-based applications concerning three key aspects: memory behavior (data locality), scheduling, and performance. With these methods, we lay the groundwork for improving runtime system, providing insight into the interplay between the schedule’s behavior, data reuse through the cache hierarchy, and the resulting performance.
Acknowledgments

I would like to take a moment to thank all the people who made this thesis possible. First of all, special thanks to my advisors David Black-Schaffer and Erik Hagersten for their extraordinary guidance and support. Also, I would like to thank my co-authors Andra Hugo and Thomas Grass for all the fun projects and discussions we shared, and to Stefanos Kaxiras and Magnus Själander for the insightful discussions.

In addition, I would like to thank all my friends in the UART group. Ricardo Alves, Moncef Mechri and Gustaf Borgström for being amazing office-mates; Kim Anh-Tran, Johan Janzén, Mehdi Alipour and Chris Sakalis for the great time during fika; and to the good (now graduated) friends that inspired me throughout the years, Andreas Sembrant, Nikos Nikoleris, Andreas Sandberg, Vasileios Spiliopoulos, Xiaoyue Pan, Muneeb Khan, and Konstantinos Koukos. Also, special thanks to Alexandra Jimboborean and Alberto Ros for the outstanding support.

Finally, I would like to thank my family, which I love wholeheartedly, for their never-ending support and love.
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.


Reprints were made with permission from the publishers.
# Contents

Part I: Introduction .................................................................................................................. 5

1 Introduction .......................................................................................................................... 7
   1.1 The Rise of Task-Based Programs ................................................................................. 7
   1.2 The Yin and Yang of Runtime Systems ........................................................................ 10

2 The Schedule-Memory-Performance Triad ......................................................................... 12
   2.1 Memory and Performance: A Well Studied Area ......................................................... 12
   2.2 Scheduling Becomes a Challenge .................................................................................. 13

3 Breaking Down the Problem ............................................................................................... 17
   3.1 Contribution 1: Task Pirating ...................................................................................... 17
   3.2 Contribution 2: StatTask ............................................................................................. 20
   3.3 Contribution 3: TaskInsight ......................................................................................... 23

4 Conclusion ............................................................................................................................. 28

References ................................................................................................................................ 29

Part II: Papers .......................................................................................................................... 33

Paper I: Shared Resource Sensitivity in Task-Based Runtime Systems ................................. 35
   I.1 Introduction .................................................................................................................... 36
   I.2 Experimental Setup ....................................................................................................... 37
   I.3 Evaluation ...................................................................................................................... 39
   I.4 Related Work ................................................................................................................ 42
   I.5 Conclusion ..................................................................................................................... 43
   I.6 Acknowledgements ....................................................................................................... 43

References ................................................................................................................................ 45

Paper II: Formalizing Data Locality in Task Parallel Applications ......................................... 47
   II.1 Introduction .................................................................................................................. 48
   II.2 Motivation: Task Data Reuse ...................................................................................... 49
   II.3 Theoretical Background .............................................................................................. 53
   II.4 Statistical Cache Modeling with Task Support ............................................................ 55
   II.5 Evaluation ................................................................................................................... 60
   II.6 Related Work .............................................................................................................. 62
II.7 Conclusion and Future Work .......................................................... 63
II.8 Appendix: Proofs ........................................................................... 64

References ............................................................................................ 67

Paper III: TaskInsight: Understanding Task Schedules Effects on Memory and Performance ................................................................. 69
  III.1 Introduction .................................................................................. 70
  III.2 Motivation .................................................................................... 72
  III.3 Through the data-reuse glass ....................................................... 74
  III.4 Analyzing Performance ................................................................. 77
  III.5 Multi-threaded Executions ............................................................. 78
  III.6 Implementation ............................................................................. 86
  III.7 Related Work ............................................................................... 87
  III.8 Conclusion ................................................................................... 88
  III.9 Acknowledgments ......................................................................... 89

References ............................................................................................ 91

Paper IV: Analyzing Performance Variation of Task Schedulers with TaskInsight .................................................................................. 93
  IV.1 Introduction .................................................................................. 94
  IV.2 Motivation .................................................................................... 96
  IV.3 Motivating Example ..................................................................... 99
  IV.4 Through the data-reuse glass ....................................................... 101
  IV.5 Analyzing Performance ................................................................. 104
  IV.6 Multi-threaded Executions ............................................................. 105
  IV.7 Detecting problems in other benchmarks .................................... 113
  IV.8 Implementation ............................................................................. 116
  IV.9 Related Work ............................................................................... 117
  IV.10 Conclusion .................................................................................. 119
  IV.11 Acknowledgments ......................................................................... 120

References ............................................................................................ 121
Part I:
Introduction
1. Introduction

1.1 The Rise of Task-Based Programs

In recent years, parallel multi-core architectures have become standard. Software developers now have to be prepared to program for these platforms, which is significantly more difficult than for single-core processors. This not only makes the programming process harder, more complex and more difficult to debug, but also means that optimizing for performance now requires the understanding of many more factors and hardware components, as well as their interactions.

As a result, different parallel programming models have been introduced, increasing the level of abstraction to simplify the reasoning, coding and debugging process while offering enough flexibility to adapt to multiple architectures. These programming models are often classified according to two key aspects: process interaction and problem decomposition [31, 32, 15].

Process interaction relates to which mechanisms the multiple parallel processes use to communicate with each other. The most common forms of interaction are shared memory and message passing, though interactions can also be invisible to the programmer or implicit (e.g., concurrent functional programming).

Shared memory is probably one of the most popular alternatives used today. While it is probably the easiest approach for reasoning, it also is an efficient way of passing data between processes: parallel processes read and write to a shared global address, asynchronously. Current multi-core processors and programming languages have support for shared memory programming models such as Pthreads, OpenMP and Cilk. A second very popular alternative within this category, particularly for scientific workloads, is message-passing, where parallel processes exchange data by passing messages to one another either synchronously or asynchronously. Some examples are D, Scala, Occam and Limbo.

On the other hand, problem decomposition relates to how the constituent processes of a parallel program are formulated, through either data-, task- or implicit parallelism. A task-parallel model is focused on processes or threads of execution, where each process performs certain operations, which might emphasize the need for communication. In a data-parallel model, a set of tasks will operate independently on a structured data set, usually on disjoint partitions. Finally, in an implicit model of parallelism nothing is revealed to the programmer: the compiler (i.e. using automatic parallelization), the
Table 1.1. Parallel programming models and classification according to process interaction and problem decomposition.

<table>
<thead>
<tr>
<th>Model, Language, Library</th>
<th>Interaction</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthreads, CUDA, OpenMP, Cilk</td>
<td>Shared Memory</td>
<td>Data</td>
</tr>
<tr>
<td>OmpSs, OpenMP tasks</td>
<td>Shared Memory</td>
<td>Task</td>
</tr>
<tr>
<td>MPI, D, Erlang, Scala, TensorFlow</td>
<td>Message Passing</td>
<td>Task</td>
</tr>
<tr>
<td>Concurrent Haskell, ML</td>
<td>Implicit</td>
<td>Task</td>
</tr>
<tr>
<td>Compiler auto-vectorization, super-scalar processors, VLIW</td>
<td>Implicit</td>
<td>Implicit</td>
</tr>
</tbody>
</table>

runtime or the hardware (i.e. super-scalar architectures) are responsible for exposing parallelism out of the application, converting sequential code into parallel code.

Table 1.1 shows a summary of current parallel programming models with their classification according to process interaction and problem decomposition. Alternatives such as Pthreads, OpenMP and MPI have been wildly adopted as they are portable to different platforms and expressive enough to implement any kind of application.

However, current and future architectures are becoming increasingly more complex: SoC designs with thousands of cores, processors with multiple CPUs and GPUs on the same die, deep multi-level cache hierarchies, different memory network topologies or memory-to-memory interconnects, etc. Furthermore, as hardware parallelism is increasing, resources such as the caches, busses and main memory are being shared in different ways. Despite libraries and models like Pthreads and MPI are adapting to these changes providing new APIs and support for the programmers, the scaling is still limited by the lack of understanding of performance bottlenecks.

To address these difficulties, task-based programming inserts an additional level of abstraction between the application and the system. In a task-based execution, the application spawns small independent tasks (functions or units of code) and submits them to a runtime system, where will be queued for execution. The runtime, executing on particular thread or core, picks the next available tasks according to a scheduling policy and assigns it to the available resources (threads or physical cores).

This paradigm has numerous advantages over other alternatives for several reasons:

1. **Tasks allow higher-level problem description:** tasks enable the programmers to think at a higher level. When using threads, for example, the application needs to be structured in terms of physical threads to yield good efficiency, as there is one logical thread per physical thread to avoid under- or over-subscription. With tasks, the programmer can
Figure 1.1. High-level overview of a task-based system. The applications interfaces with a runtime system to submit tasks (A to E) using an abstract model. The runtime, aware of both the architecture and information of the application, will determine the execution order (schedule) of the tasks.

focus on the dependencies between tasks, leaving the scheduling to the runtime scheduler.

2. **Tasks are lightweight**: tasks are much lighter weight than threads, and current implementations and frameworks reported speeds up to 20 times faster when starting and terminating tasks.

3. **Tasks can be scheduled for performance**: thread schedulers usually distribute time slices in a round-robin fashion, in a *fair* distribution, because it is the safest strategy when the scheduler is oblivious of the application structure. However, in a task-based program, the task scheduler has higher-level information that could use to guide the scheduling, trading off fairness for performance.

4. **Tasks simplify load balancing**: in addition, the runtime scheduler takes care of the load balancing, using the right number of threads and distributing work evenly across physical threads.

These strengths have led to the development of many production-quality frameworks, including OmpSs [10], OpenMP tasks [62], Intel’s TBB and StarPU [47]. Figure 1.1 shows how these systems are typically organized internally. The application interfaces with the runtime system in a high-level way, removing the architectural details from the program logic. This allows the system to optimize the execution for better performance using both program and architecture information, but in a transparent way for the programmer.
1.2 The *Yin* and *Yang* of Runtime Systems

Delegating the execution to the runtime not only simplifies the coding process, but also provides flexibility to adapt to multiple architectures in a manner transparent to the programmer. Having a runtime also allows optimizations for different executions. The same application could achieve different performance or energy savings depending on the goal of the runtime and how it is configured. In addition, the runtime layer also provides an opportunity to improve application efficiency without changing the application itself, by introducing enhanced runtime techniques.

On the other hand, developing a runtime system demands full attention to several aspects. First of all, adding an extra layer between the application and the Operating System creates an *overhead* during the execution. In order for this approach to optimize for performance, the overhead of the runtime needs to be negligible, or at least, compensated by the overall performance achieved and total execution time of the application.

Second, runtimes’ interfaces should be expressive enough to cover a wide range of parallel applications, from a problem decomposition perspective (e.g., functions should be expressed as independent tasks).

Lastly, with a task-based programming model, the programmer gives up fine-grained control of the parallelism and scheduling to the runtime. For this to be a good trade-off, the runtime needs to do an efficient job during the execution, at least as efficient as what the programmer would specify.

In order to build better runtime systems, it is essential to understand (1) how sharing resources affect the performance of the executing tasks, (2) how does...
the runtime scheduler affect the memory behavior of the application, and (3) how the interplay between the schedule chosen and the application’s memory behavior affects the performance of the execution.

We summarize this into what we call the Scheduling-Memory-Performance triad, shown in Figure 1.2. In this thesis, we present techniques and models to understand how these three key factors interrelate to each other during the execution. More specifically, we study how the data locality properties of the applications are affected by scheduling, and therefore, how data is placed and reused through the caches affecting the performance. This allows to explain performance variation across different schedules for the same application. These methods set up a unique platform to reveal insight into how to improve the performance of large-scale task-based applications.
2. The Schedule-Memory-Performance Triad

The Schedule-Memory-Performance triad (Figure 1.2), summarizes the three key components of the execution of task-based programs.

Previous work (Section 2.1) has covered the interactions between memory and performance extensively. It has been shown how improving the application’s data locality by reusing data more efficiently had a positive impact on performance [49, 9, 33, 19, 34, 26]. Furthermore, several techniques have been proposed to understand performance changes caused by how data is shared in parallel applications [49, 57, 48].

However, compared to previous programming models, task-based applications introduce scheduling through the runtime system. Both other components (memory and performance) are affected by scheduling, creating interactions in two new directions: how does scheduling relate to performance, and how does scheduling relate to data locality.

Since the interactions between the three factors are very complex, we will break them down to be able to understand them better. To do that, we will start by exploring how previous state-of-the-art contributions provide insight into this interplay, and how they are limited in a task-based context.

2.1 Memory and Performance: A Well Studied Area

The interactions between memory and performance have been studied in-depth. When we consider sequential or thread-based applications, there is a direct correlation between its data locality properties and its achieved performance. Applications that expose more temporal or spatial locality in their memory access patterns are able to serve more of them from higher levels in the memory hierarchy (caches), which are much faster than main memory.

Previous work has explored these effects in depth in several ways. One common strategy is to change the software and execute on real hardware, observing performance differences. This empirical evaluation often yields good tuning and optimization results, but it is a long process which does not provide general insight applicable to other situations.

On the other hand, simulation is able to replicate both the applications’ and hardware’s behavior across varying configurations. Among other things, this enables the programmers to reason about their applications, and their sensitivities to different hardware configurations. Even though detailed architectural simulation [5, 17] is a common approach for evaluating architectural changes
and new designs, it is too slow for understanding applications. As a result, recent work has developed improvements such as near-native execution simulation [29] using smart warming heuristics [6, 27], as well as high-level simulators [8, 30].

An alternative to simulation is Memory Modeling, and it has been successful as a way to characterize data locality based on architecturally-independent information. One example within this category are statistical cache models [49, 57, 48, 56] where the applications are profiled to collect information about their memory accesses, which can be used to predict cache behavior for arbitrary cache sizes. These techniques are very attractive to study the applications, as they are flexible enough to model a wide range of cache sizes, while being fast and low overhead.

In addition, there has been significant work in analytical models to understand and model performance changes based on applications’ characteristics. Analytical performance models [7, 2, 16, 20] are able to predict the variation in performance (in cycles) for arbitrary architectural characteristics. This has been extremely useful to identify how applications respond to different memory hierarchy designs and core configurations, in a quick and accurate way. Another example is the analysis of sensitivity to resources such as caches [11, 25] and bandwidth [14, 24, 23].

Despite there has been extensive work looking at how memory and performance interrelate, this work has generally assumed a fixed program schedule which does not hold for task-based programs.

2.2 Scheduling Becomes a Challenge

Unlike sequential programs, the execution schedule in task-based applications can change from execution to execution. This is due to the fact that scheduling is non-deterministic. The execution order will be determined according to a scheduling policy, based on the pool of available tasks at a given point in time.

Since each task performs different operations, having different execution orders means that the applications’ overall behavior may vary. Let us examine the code shown in Figure 2.1. Function A, after operating on the input data, calls two independent functions, B and C, which also work on the input data. After both functions are done, a third function D will use the result from B and C, along with the original input data, to produce the final result. This is a common code structure for many applications such as numeric solvers and image processing applications.

It is possible to reason about this structure as a dependency graph, shown in Figure 2.2. We can see how first A needs to be performed, later B and C, which are independent from each other (performing a different operation), and finally D combining the results. By looking at this structure we can immediately start
recognizing the inherent parallelism of the application, as well as the internal structure, inherent to how it was programmed.

If we consider the code in Figure 2.1 to be a sequential application, the execution will consist of functions A, B, C and D executed in that order, unless the code is re-written. This is shown in Figure 2.3 (top) where we can see how the functions are executed over time.

On the other hand, if the sample application is structured as tasks, the application will start submitting tasks A, B, C and D to the runtime system. As the runtime sees available tasks in the task pool, it picks some of them for execution (for example starting with A). Once A is done, both tasks B and C are ready to be executed, and the runtime will make a choice depending on its scheduling policy. This allows either B or C to be executed next, without requiring any change to the application.

This flexibility during the execution becomes particularly interesting when considering how tasks interact through the memory system. This is illustrated in Figure 2.3 (bottom) by two different schedules (Schedule 1 and Schedule 2), where each function is a task. The function D (now task D), uses both the results from tasks B and C, which take roughly the same amount of time to execute. However, D uses a much larger portion of data from C’s output than from B’s output. This difference in how they use data means that the schedules’ interactions through the memory system will result in significantly different performance, even though they are logically equivalent.
Figure 2.3. Scheduling as a challenge. In sequential applications, program order is defined beforehand. In task-based applications, the tasks execution order is decided by the scheduler during runtime, allowing different executions for the same application. Changing the execution order affects memory behavior, which may change the overall performance.

Let us consider that the functions B and C generate the same amount of data, and that this application is executing in a system with a last-level cache large enough to hold either B’s or C’s output data, but not both. If the runtime chooses Schedule 1, when task D starts, all accesses to C’s data will hit in the cache, since C was just executed. On the other hand, if Schedule 2 is chosen, B will evict all of C’s data when executing, making D miss on every access to C’s output. D will only be able to serve accesses to B’s output. Given than D uses significantly more data from C than from B, its performance is affected by the increase in cache misses.

These cache misses are caused by the runtime scheduler’s reordering of the tasks in a memory-system oblivious manner. The result of this can be a slowdown in the execution of D, as shown in Figure 2.3.

As we have seen in the example, changes in the runtime schedule can result in significant differences in performance due to the interactions through the memory system, despite producing equivalent results.
While the interactions between the scheduler and memory system can have a significant impact on performance, there is a significant lack of tools to understand the reasons behind it, providing insight into this interplay. This limits the development of better runtime systems that are able to trade off some schedules for others that maximize performance under specific memory systems and platforms. An example of how the runtime’s data placement can be improved by providing extra information about the application can be seen in [22]. However, we are still not able to provide to the runtime useful information about the effects of scheduling in a flexible manner.

In this thesis, we address this lack of tools by breaking down the different interactions into smaller problems, and extending existing techniques to these new context. In the following section, we summarize our contributions, including which problems are addressed and how they relate to the scheduling-memory-performance triad: scheduling to performance interactions (Contribution I), scheduling to memory interactions (Contribution II) and finally, the interplay between the three factors (Contribution III).
3. Breaking Down the Problem

As we have seen previously, many factors affect the execution of task-based applications. Different task schedules can change the overall performance of the programs due to changes in how tasks interact through the memory system. To understand how using memory differently has an impact on performance, and the role of different scheduling decisions on this impact, we start by looking at specific aspects of the triad.

We present three techniques (Papers I, II and II) that are summarized in Figure 3.1. Each of the techniques covers one particular area of the Scheduling-Memory-Performance triad. First, we focus on how the performance of tasks changes when sharing resources, which is an interaction between scheduling and performance. Second, we study how different schedules expose different data reuse patterns, which is an interaction between scheduling and memory (data locality). Finally, we connect all three parts of the triad (Memory, Scheduling and Performance) by presenting a technique to link changes in memory behavior caused by scheduling to the performance of the execution.

3.1 Contribution 1: Task Pirating

The first area we explore is how tasks behave when sharing the cache. This is an interaction between scheduling and performance, and allows us to evaluate
Figure 3.2. Different sensitivities to cache sharing. In (a) Task is running in isolation, fully utilizing the cache. In (b) tasks B and C share the cache evenly. In (c), task D has a more aggressive memory access pattern, reducing the space that task B gets in the cache. In (d), task D is combined with a non-memory intensive task E, having a symbiotic relationship at the memory system level.

how sensitive is the performance of the tasks to the shared cache when co-executing.

When a task is running in isolation, it can fully utilize the resources, such as the shared and busses to move data from memory. On the other hand, when running in parallel, tasks will fight for many of those shared resources, and especially for the last level cache which is crucial for performance. However, tasks’ sensitivity to resource sharing varies with the task type, meaning that not all of them will be affected the same.

This is illustrated in Figure 3.2 with several examples. Case (a) shows a single task in isolation, with all private and shared caches available to itself. When co-running, several scenarios may occur. In (b), tasks B and C have a similar memory behavior, and in practice, resources will be shared evenly, having 50% of the shared cache for each. In (c), task D has a very aggressive memory access pattern, and it is able to fill most of the last level cache, reducing the space that task B gets in the cache. This might degrade the performance of B, as it will miss more in the cache. Finally, in (d) we can see how the memory requirements of E complement in a symbiotic relationship with D’s memory behavior, creating a nice interaction and maximizing resource usage.

All these cases will have different implications on the performance achieved by these tasks, and it is up to the scheduler to determine whether is good to co-execute tasks and how. In Paper I, we leverage Cache Pirating [11] to study cache sensitivity in the context of tasks.

In Cache Pirating the application subject to study is co-executed multiple times, from start to finish, with a pirate application. For each execution, the Pirate will steal a certain amount of the shared cache by issuing memory requests.
Figure 3.3. Task Pirating. A Cache Pirate is co-executed with the tasks applying different pressure levels. Tasks gradually get less space in the cache. At the same time, statistics from hardware performance counters are collected in a per task basis to construct sensitivity curves.

At a certain rate. At the same time, the pirate will read and save information from the hardware performance counters of the target application in order to be able to compute its performance (typically cycles per instruction (CPI) and cache miss ratio).

In addition, comparing the performance information to the cache miss ratios allows us to understand how sensitive the program is to a particular cache pressure: the more the application uses the shared cache, the more its performance is likely to suffer when applying pressure with the pirate. From Cache Pirating it is possible to get cache miss ratio and CPI curves for all the different pressure levels. Sensitive applications will experience lower CPI when miss ratio is higher, while non-sensitive applications will show negligible difference.

Having this sensitivity information is crucial to optimize the performance during the execution: sensitive tasks or threads should not be co-executed with other cache-hungry threads (if possible), or their execution will slow down. However, using the Cache Pirating technique out of the box with a task-based application would give us useful information about the overall sensitivity to the shared cache, but would not provide the per-task information needed to optimize the scheduling.

To address this, in Paper I we extend Cache Pirating to a technique called Task Pirating. Figure 3.3 shows an overview of the methodology. Compared to [11], our approach puts pressure using a pirate application, while recording statistics per task (hardware performance counters). This is done by interfacing with the runtime so that hardware performance counters are read at the beginning and end of each task, in comparison of fixed intervals defined by the original Cache Pirate. In this way, we can understand the per-task shared sensitivity.
One advantage of the Task Pirate is that each individual task is evaluated based on how much their performance changes when sharing the cache. Since each task has a different task type, it is possible to compute the average task behavior per type.

The Task Pirate runs with the application through the entire execution, and the data collection of the sensitivities is done during runtime, while the analysis (computation of miss ratio and CPI curves) is done offline. The overhead of the technique is negligible, as with the original Cache Pirate methodology. The method is input-specific, meaning that a change in the input dataset requires a re-execution of the Task Pirate. However, once the miss ratio and CPI curves are constructed, this information can be saved and used at runtime.

Our findings show that reasoning on a task-type basis allows to draw conclusions valid for many tasks: a whole group of tasks can suffer significantly when it is combined with a cache-hungry task of a different type (e.g., tasks B and D in Figure 3.2). Analogously, some combination of tasks (e.g., tasks of type D and type E in Figure 3.2) have a symbiotic relationship at the shared-cache, making them a good fit for co-executing regardless of which task instances are.

This information enables the scheduler to make better decisions at runtime, as there are usually only a handful of task types per application (up to 15 in our studied applications), compared to the number of task instances (up to 50k).

3.2 Contribution 2: StatTask

Performance variation due to scheduling does not only come from co-scheduling tasks sensitive to cache sharing (i.e. sharing data at the same time, or temporal locality). The example depicted in Figure 2.3 shows how reusing data over time differently can hurt the overall performance (i.e. using data left in the cache, or spatial locality).

The fundamental reason behind this is a change in the data locality of the application caused by the schedule: depending on the execution order chosen by the scheduler, later tasks may find their data has been evicted from the cache due to other tasks executed before them, thereby reducing cache hits and performance.

A widely adopted method to study locality properties are Statistical Cache Models [49], due to their speed and ability to model different cache sizes without requiring additional information.

Two popular examples are StatCache [49] (for random replacement caches) and StatStack [57] (for LRU caches), which work as follows: the target application is executed along with a profiler that captures high-level information about its memory accesses, such as the address, type of operation (read or write), program counter, etc. To reduce the overhead, only a small portion of the memory accesses are sampled (one every 100 thousand or more), and
observed until they are reused, allowing the profiling phase to incur very low overhead.

The sampled data reuse information can then be used to model the cache behavior. First, the *reuse distances* are computed, i.e. the number of intervening memory accesses between each reuse, and a Reuse Distance Histogram (RDH) is calculated. By looking at the RDH, it is possible to identify how the reuses are distributed. Furthermore, for a given cache size, it is possible to determine the maximum reuse distance allowed before the data is evicted. This enables modeling of different cache sizes because determining if a data reuse will be a hit or a miss in the cache boils down to whether the reuse distance is less or greater than a particular threshold, given by the cache size.

However, the execution of a task-based program changes drastically depending on how the runtime system schedules the tasks. If we profile a particular schedule with StatTask, we would identify data reuses and certain reuse distances. Since the order of the tasks changes for a different schedule, their memory accesses also change, as well as when those data reuses happen. Profiling a second time may result in a completely different reuse distance distribution, and thus, different conclusion about the locality properties for the same application.

Figure 3.4 illustrates this situation. On the top, we can see a sequential application, how its memory accesses are sampled, and how reuse distances are identified over time with StatStack. On the bottom, we see the a task-based application consisting of tasks A, B and C with different schedules. In Schedule 1, Task B is reusing data from A with a reuse distance of 5. However, in Schedule 2 task C is scheduled between A and B. Task B still reuses data from A, but now the distance is 15, due to the memory accesses generated by C happening between A and B.

Proposed extensions [28, 21, 35] to study data reuse of task parallel runtimes based on reuse distances can characterize, holistically, the data locality properties of the applications. However, these techniques are not flexible enough to predict locality for arbitrary schedules.

In Paper II, we extended the StatCache and StatStack models to be used with task-based applications by introducing the StatTask model. StatTask allows us predict cache behavior for any schedule from a single profiling run, maintaining the accuracy and low-overhead benefits from previous statistical cache models. The model focuses on the study of the temporal locality of the schedules (i.e., how tasks reuse data through the private caches over time). However, it can also be used to analyze spatial locality (i.e., how tasks reuse data at the shared cache when co-running), and thus, it is complementary to the Task Pirate (Contribution 1).

StatTask profiles a single schedule of the target application. When memory accesses are collected, information from the task originating is also saved along with the address, the type and program counter. Compared to previous models, this information allow us to classify data reuses based on the tasks
Figure 3.4. StatTask Problem: In task-based applications, changing the schedule (execution order of tasks) changes the way data is reuse throughout the execution, and thus their reuse distances, which is a challenge for existing statistical cache models.

involved. A reuse that only happens within one task is a *private* reuse. Analogously, if two tasks are involved it is a *shared* reuse.

Later, the Reuse Distance Distribution is built for the profiled schedule, and finally, cache miss ratio curves are computed. Moreover, if the cache miss ratio curve is desired for a different schedule, StatTask can recompute the reuse distances from the previously captured reuses by looking at the classification (private vs. shared), which avoids the need for re-profiling every possible schedule. This is shown in Figure 3.5: using traditional statistical cache models would require a profiling phase for each schedule of the same application, while with the StatTask model only a single profiling phases is needed.

In our studies, we demonstrate the potential of StatTask’s analysis to understand task-based scheduling by examining applications from the BOTs benchmarks suite. We show that a range of applications have potential to share 35% of the memory accesses between tasks on average (up to 80%). We also demonstrate how this method can be used to better understand the sharing characteristics. With StatTask we have a new ability to rapidly explore the impact
of task scheduling on cache behavior, which opens up a range of possibilities for intelligent, reuse-aware schedulers and better performance.

3.3 Contribution 3: TaskInsight

With Papers I and II, we have provided new methods that give insight into (1) how the performance of the tasks changes when they are co-scheduled, and (2) how the temporal locality of the application is affected by the schedule. Making the runtime aware of these is a key step to better informed scheduling decisions. However, we have not yet explored one missing link between the three factors: scheduling, memory and performance.

With Papers III and IV, we explore the following fundamental question: How does the performance of the application relate to a change in memory behavior caused by scheduling?

Runtimes try to be entirely automatic, but expose some parameters to the user to guide the execution. In many cases, this is useful to tune particular applications for specific inputs. With the increasing complexity of these systems, it is becoming more and more difficult for the programmers to set these parameters for an efficient execution, leading to degraded performance. As a result, significant work has been done to develop runtime systems with better scheduling heuristics. For example, there are numerous scheduling policies that optimize for load balancing (i.e. work stealing) but they are unaware of data locality [1], which often causes worse performance on memory-bound applications.

Generally, developers attempt to characterize their workload based on data reuse without considering the dynamic interaction between the scheduler and the caches. This is simply because there has been no way to obtain precise information on how the data was reused through the execution of an application, such as how long it remained in the caches, and how the scheduling decisions influenced the reuse history. Without an automatic tool capable of providing insight as to whether and where the scheduler misbehaved, the
programmer must rely primarily on intuition, interactive visualization of the execution trace [18, 85, 80] or simulating the tasks execution in a controlled environment [92, 83] to understand and adjust the scheduler for improved performance.

In Papers III and IV we present TaskInsight, which is a new method to characterize the scheduling process quantitatively. The method was formulated to address three questions that are key to understand the performance of a particular schedule, and thereby the scheduler itself:

1. **What** scheduling decisions impacted the performance of the execution?
2. **When** were those decisions taken?
3. **Why** did those decisions affect the performance?

TaskInsight shows how the data reuse between tasks can provide vital information for answering these questions, as they can be quantified over time, exposing the interactions between the tasks’ performance and their schedule. Further, TaskInsight can interface directly with the runtime system to provide this information both to the programmer and the scheduler.

An overview of TaskInsight is shown in Figure 3.6. The technique consists of two phases: *profiling* and *instrumentation*. During profiling, a profiler captures and saves information about the memory accesses of the target appli-
cation for a particular schedule, including unique task IDs (as in Paper II with StatTask).

Later, in the instrumentation phase, the application is executed a second time with the same schedule. As in Paper I, while the application is executing, hardware performance counters, such as instruction counters, cycles, cache accesses and misses, are read at the beginning and end of each task. Reading the hardware performance counters at these specific points (start and end of a task) not only allows to study the behavior per task, but also avoids noise (unwanted accesses and cycles) added by the runtime when there is no task executing. Avoiding runtime noise is mandatory to understand the fundamental impact on the application’s performance when changing the schedule.

It is also worth noting that TaskInsight requires two executions with the same schedule because the profiler is implemented using binary instrumentation, which adds an overhead to the execution, affecting the native performance of the application. If we read the hardware performance counters while the profiler is attached, the results would be affected. Instead, TaskInsight executes the application a second time, without the profiler. However, this limitation is not intrinsic to the technique: if the profiler is sufficiently low-overhead, the two steps could be combined.

After profiling and instrumentation, saved memory access are analyzed to differentiate private and shared data per task. For a given schedule, TaskInsight classifies the memory accesses issued by each task into one of the following categories:

- **new-data**: If the memory address is used for the first time in the application.
- **last-reuse**: If the memory address was used by the previous task.
• **2nd-last-reuse**: If the memory address was used by the second-to-last.

• **older-reuse**: If the memory address was used before, but by an older task.

This is shown in Figure 3.6 as the Data Classification step. Later, in the Analysis Over Time step, the previous classification is displayed over time, following the execution order of the tasks, and combined with performance metrics obtained from the readings of the hardware performance counters. By repeating the process for different schedules, it is possible to understand *when* and *where* performance variation is connected to changes in memory behavior.

An example of the analysis shown by TaskInsight is shown in Figure 3.7, for the histogram application implemented using the OmpSs runtime. For two different schedules, *smart* (wf policy in OmpSs) and *default* (default OmpSs scheduler) \(^1\), the data classification is connected to the performance information. This allows us to detect particular tasks that had a performance degradation, when they were executed, and if the reason behind the degradation is reusing old data no longer in the cache. Figure 3.7 shows how scheduling tasks that bring significant new data in the middle of the execution (Naive schedule, task 18) can hurt the overall performance due to more L2 cache misses.

In case we want to analyze multiple schedules, the instrumentation phase needs to be executed for each of them. On the other hand, the profiling phase is only needed once for any arbitrary schedule as shown in Figure 3.8. This is one of the main advantages of TaskInsight as a low-overhead technique, as profiled data can be saved and reused to model any other schedule.

In Papers III and IV, we study a broad range of applications. We show how TaskInsight not only shows per-task performance, but also provides an explanation for why tasks of the same type can have significant variation in performance (up to 60% in our examples). Our findings show how programmers can now quantitatively analyze the behavior of the scheduling algorithm and the runtime can use this information to dynamically make better decisions for both sequential and native multi-threaded executions. Our analysis exposed scheduler-induced performance differences of above 10% due to 20% changes in data reuse through the private caches and up to 80% difference data reuse through the shared last level cache.

Overall, TaskInsight allows us to understand the impact of scheduling changes in a way that can be used by schedulers to improve performance, increasing reuse through the caches. Approaches that only measured the actual cache miss ratios per task (e.g., using hardware performance counters) are unable to

\(^1\)In Papers III and IV, the default scheduling policy is named *naive*. As seen in ??, the *default* scheduling policy is uses a single/global ready queue. Tasks with no dependencies are placed in this queue and are executed in a FIFO (First In First Out) order. The *wf* policy implements a local ready queue per thread. Once a task is finished the thread continues with the next task created by this later. The main difference between them is the locality optimization where *wf* prioritizes the reuse between the tasks.
Figure 3.8. Analyzing multiple schedules with TaskInsight. The instrumentation phase has to be executed for each schedule to collect hardware performance counters statistics, but only one profiling phase is needed.

trace back changes in memory behavior to the scheduling decision that caused them in this manner. As a result, this novel methodology enables scheduler designers to gain insight into how specific scheduling decisions impact later tasks.
4. Conclusion

Hardware on modern CPUs is becoming increasingly more complex, with deep memory hierarchies and dozens of cores. Task-based programming stands out as a simple and flexible option for parallel programming. It recently gained popularity among developers because it allows to delegate many intricate decisions to a runtime system, such as scheduling tasks for execution as well as placing and moving data.

For runtimes to optimize performance, it is necessary to understand three key components: **scheduling**, **data locality** (memory behavior) and **performance**, and how they affect each other. However, while current approaches may give an accurate view of the application on one particular system, they are oblivious to different schedules. Therefore, these approaches are not appropriate to fully understand the interplay between all the components of task-based executions.

In this thesis, we addressed the lack of tools to analyze these interactions by introducing new techniques and models to understand how the performance of the tasks is affected by scheduling (Paper I), how the data locality of the tasks changes with the schedule (Paper II), and finally, how the performance of the application was affected by changes in the memory behavior of the tasks because of scheduling (Papers III and IV). This allows to study the relationships between the three factors in a holistic way, setting up a unique platform to reveal insight into how to improve the performance of large-scale task-based applications.
References


[21] Hao Luo, Xiaoya Xiang, and Chen Ding. Characterizing active data shar-
ing in threaded applications using shared footprint.


[34] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm.

Part II:
Papers
Paper I
Paper I

Shared Resource Sensitivity in Task-Based Runtime Systems

Germán Ceballos and David Black-Schaffer

In Proceedings of the
Nordic Workshop on Multicore Computing (MCC)
Halmstad, Sweden, November 2013
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.

I.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 [41]. This reduced overhead allows for much finer-grain parallelism.

Popular implementations include Intel’s Thread Building Blocks [41], OmpSs [43] and StarPU [36]. 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 [36]. 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 [40], 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.

I.2 Experimental Setup

I.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 [40] 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. Sec-
ond, 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.

I.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 \cite{parsec} 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.

I.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-rank 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 \cite{papi} 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 ap-
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.

I.3 Evaluation

I.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 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 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.

I.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.

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.

I.4 Related Work

Yoo [44] presented work on resource-contention aware scheduling techniques, exploiting data locality at scheduling time, and techniques for modeling locality patterns. Bhaduria et. al. [37] and Blagodurov et. al. [39] 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 [36] that were later used to design the StarPU scheduler.
I.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.

I.6 Acknowledgements

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.
References


Paper II
Paper II

Formalizing Data Locality in Task Parallel Applications

Germán Ceballos, Erik Hagersten and David Black-Schaffer

In Proceedings of the

International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)
Granada, Spain, December 2016
Abstract

Task-based programming provides programmers with an intuitive abstraction to express parallelism, and runtimes with the flexibility to adapt the schedule and load-balancing to the hardware.

Although many profiling tools have been developed to understand these characteristics, the interplay between task scheduling and data reuse in the cache hierarchy has not been explored. These interactions are particularly intriguing due to the flexibility task-based runtimes have in scheduling tasks, which may allow them to improve cache behavior.

This work presents StatTask, a novel statistical cache model that can predict cache behavior for arbitrary task schedules and cache sizes from a single execution, without programmer annotations. StatTask enables fast and accurate modeling of data locality in task-based applications for the first time. We demonstrate the potential of this new analysis to scheduling by examining applications from the BOTS benchmarks suite, and identifying several important opportunities for reuse-aware scheduling.

II.1 Introduction

Multicore architectures bring new levels of performance, but at the cost of more complex development and performance analysis, particularly with regards to shared memory system resources, such as caches. Multicores have traditionally been used with multi-program or thread-based workloads, and many tools exist for analyzing their performance.

Recently, task-based programming has become popular as tasks are simple to program and move the complexity of parallel scheduling and load balancing to a runtime, instead of requiring explicit user-directed threading. As a result, task-based programs can more easily take advantage of available resources. These strengths have led to the development of many production-quality frameworks, including OpenMP tasks [62], Intel’s TBB and StarPU [47].

However, understanding performance of these systems has become harder due to shared resources, demanding new analyses. Existing tools analyze scheduling and load-balancing [60, 54, 63], but have largely ignored shared data reuse between tasks. This is a substantial omission as data locality (through caches) is one of the key factors for performance on today’s multicore processors. Understanding the reuse of shared data between tasks is critical for optimizing cache locality and memory behavior.

For serial or thread-based applications, there has been an extensive work on characterizing data reuse, including instrumentation [64] and statistical modeling [49, 57]. However, these methods only provide insight into the memory behavior of the observed execution of the program. Yet a key strength of task-based programs is that they have great flexibility in how they schedule work (tasks), as they can do better load balancing and parallelism exploitation. However, this affects the actual data reuse.

This paper presents a novel statistical cache model (StatTask) that enables analysis of cache behavior. The model captures the interaction of shared data
Figure 6. StatTask allows us to re-order profiled reuse pairs to obtain the profile for an arbitrary schedule without having to execute and profile the schedule explicitly.

between (and within) tasks and is capable of estimating how data reuse and locality is affected by scheduling. With this, it can predict cache behavior for different execution schedules from the analysis of a single run.

Figure 6 illustrates the potential of having such a model. While typical statistical cache modeling approaches would require one profiling phase per application schedule, StatTask allows us to model any arbitrary schedule from a single profiling run.

To understand the potential of task-based data reuse analysis, we first analyze the overall data reuse in the BOTS task benchmark suite (Section II.2). We then look at sparseLU in detail, by connecting its reuse profile and schedule. With that background, we introduce a detailed formal description of the underlying data representation for statistical cache frameworks (Section II.3), and our contributions to handle tasks and schedules (Section II.4). Finally we describe the details of our implementation and analyze the results obtained when applying it to multiple benchmarks of the BOTS suite with different input data sets and schedules (Section II.5). In addition, we discuss this work in relation to existing profiling tools (Section II.6), addressing future work and limitations.

II.2 Motivation: Task Data Reuse

Optimizing a program for data reuse through the cache is essential for performance on modern systems: cache latency is between 4 to 30 cycles, while main memory can easily have a latency over 100 cycles. However, caches are much smaller, so data placement has to be done carefully. For task-based programs, data is initially brought into the cache by a task, and, if it is reused, it can either be from the same task (private reuse) or by a later task (shared reuse). Tasks that execute between tasks with shared reuses also bring data into the cache, which may evict the shared data, thereby turning cache hits into misses, and hurting performance.
Following this intuition, we classify accesses in task-based applications into three types, depending on who originates the access, and whether they are in the cache:

- **DRAM Accesses**: accesses that miss in the cache and must be brought in from DRAM, such as the first accesses to data.
- **Private Reuses**: accesses to data previously loaded by the same task. These will hit in the cache if it is large enough to hold the task’s entire data set.
- **Shared Reuses**: accesses to data previously loaded by another task. These will hit in the cache if it is large enough to hold the data sets of both sharing tasks, and the data is not evicted between the two tasks that share the reuse.

A scheduler that understands how its decisions affect the amount of shared reuses that hit in the cache can intelligently make a trade-off between cache behavior and other metrics. The potential for such optimizations varies across applications (depending on the amount of shared reuses and the scheduling flexibility) and architectures (depending on the sizes and configurations of the caches). To illustrate this, we analyzed a range of task-based applications from the BOTS benchmarks suite [55].

Figure 7 shows the breakdown of the accesses types. This data is schedule-agnostic, and shows the maximum potential shared reuse across all tasks. The benchmarks demonstrate significant diversity, with sort, fft, sparseLU and nqueens exhibiting over 30% shared reuses on average, and floorplan reaching nearly 80%. As the portion of shared reuses increases, the potential benefits of scheduling to ensure that those reuses are present in the cache increases as well.
If we look more closely at the sparseLU benchmark, we can see that it has a complex internal task structure, generating over 40,000 tasks when running with default settings. A subset of the tasks are shown in Figure 8. Each node is a task instance with color indicating the type and its unique id inside. Edges between tasks show the amount of shared data between them and self-edges the amount of self reuses. Importantly, this reuse graph is a property inherent to the application, and is independent of the execution.

Different execution orders will only affect how many of the shared reuses become cache hits. The effect of this is shown in Figure 10 which illustrates how execution schedules can impact cache behavior.

The schedules $S_{GOOD_1}$ and $S_{BAD_1}$ are for single-core execution, while $S_{GOOD_2}$ and $S_{BAD_2}$ are for dual-core parallel execution. The black arrows show the shared reuses that hit in the cache and the red arrows show cache misses when data comes from main memory (DRAM).

Some DRAM accesses that could have been cache accesses with better schedules are indicated with the red circles (A) and (B).

To analyze how schedules interact with the cache, we must specify the cache size to know how much data it can hold. For this examples, we assume a 1MB private cache for each core. Note that in Figure 8 the green tasks

---

**Figure 8.** SparseLU Reuse Graph, showing inter- and intra-task accesses.
(2 through 36) and the blue tasks (37 to 52) share data, but the red and green datasets are independent from each other.

If we look at the bad single core schedule ($S_{BAD_1}$), task 1 will bring 1MB of data from DRAM and store it in the cache. When it finishes, task 2 will reuse all of task 1’s data from the cache and generate 1MB of output data that will be stored in the cache. However, when task 37 is executed next according to the schedule, it will bring 1MB of its own data into the cache, evicting all of task 2’s output data. When task 3 executes after task 37, it will not find task 2’s output data in the cache, and will instead be forced to reload it from DRAM, thereby evicting all of task 37’s data. This schedule is far from optimal as task it evicts task 1’s output data from the cache that otherwise could have been reused by task 3. $S_{GOOD_1}$ avoids these unnecessary cache evictions by executing first tasks 2 through 36 and then 37 through 52, thereby maximizing the shared reuses between tasks through the cache.

These two schedules have significantly different cache miss ratios due to the execution order of the tasks. $S_{BAD_1}$ will miss approximately 1K times between tasks 1 and 52, while each task has around 12K self reuses that will hit in the cache, giving a miss ratio around 8.0%. $S_{GOOD_1}$’s better schedule, however, results in a cache miss ratio of only 0.3%.

The optimization potential is even clearer for multicore execution. $S_{BAD_2}$ shows the analogous scenario for parallel execution across multiple cores with private caches. $S_{BAD_2}$ executes the first schedulable task as soon as a core is ready. While this approach helps load balance across cores, it provides poor data locality, causing constant evictions and misses. $S_{GOOD_2}$ demonstrates a
better schedule that preserves locality by executing tasks locally to the core that spawned them, and thereby reducing cache misses by 200x from 8% to 0.04%.

This work presents a framework to understand these types of inter-task memory behavior formally for the first time. To do so, we leverage a range of low-overhead statistical cache modeling techniques (StatCache [49] and StatStack [57]) to provide statistical cache inference with sparsely-sampled memory access profiles for predicting cache miss ratios. While these existing models provide insight into memory reuse, they assume that the application execution order does not change from how it was sampled, which makes them too inflexible for task-based application analysis. Our new StatTask model addresses these shortcomings to enable fast, accurate analysis of task-based applications.

II.3 Theoretical Background
Existing statistical cache models [49, 50] model the execution of an application as a sequence of memory accesses. Although this is a widely-used formalization, it is not expressive enough to identify the tasks executing those memory accesses. To address this, we present an extended representation, which models the execution of an application as a sequence of tasks. This enables us to model different schedules from one particular set of profiled data. To explain this representation, we present the first formal description of existing statistical cache modeling and extend it to support parallel tasks.

II.3.1 Sequential Memory Access Execution Model
For statistical cache models, the execution of an application $P$ is captured as a sequence of $n$ memory accesses $E = x_1 x_2 \ldots x_n$ (the access trace of $P$). Formally, a memory access $x$ is an $m$-tuple $x = (a, p_1, \ldots, p_{m-1})$ with $a$ the memory address accessed, and each $p_j$ a different property of the access. Many previous works have used a 4-tuple $x = (a, t, p, o)$ with $t$ the time of the access, $p$ the instruction pointer and $o$ the type of operation (read, write, read-write). We will denote the address of an access $x_i$ with $a^i$.

The set of accesses of an execution $E$ will be given by $A_E = \bigcup_{i=1}^{n} \{x_i\}$. We will assume that the applications are deterministic for the inputs, making this set an inherent property (i.e. schedule and cache hierarchy independent). Note that we evaluate this assumption’s effects for different inputs in Section II.5.

II.3.2 Distances and Reuses
The access distance between two accesses $x_j$ and $x_k$ is given by $\delta(x_j, x_k) = |j - k|$. Intuitively, a memory address accessed throughout an execution is
reused when it is accessed several times. An address is reused within the cache only if there are few enough unique accesses in between the reuse such that it is not evicted from the cache before it can be reused. The set of reuses for an execution $E$ of an application $P$ will be denoted $\mathcal{R}_E^P$. If a reuse is consecutive, it implies that all addresses accessed in-between are different from the one being reused. The subset of consecutive reuses is formally defined as

$$\mathcal{R}_E^P = \{(x_j,x_k) \in \mathcal{R}_P^E | \forall x_s \in E, k < s < j, a^s \neq a^k\},$$

and it is the basic information used by existing statistical cache models to understand locality properties of an application$^2$. In this context, the reuse distance will be the distance $\delta_{\mathcal{R}_E^P}$ of the consecutive reuses.

The global set of reuses of an application ($\mathcal{R}_E^P$) is almost execution-independent: for any execution, the global set of reuses will be the same, except for the order of the pairs (tuples). This property is expressed formally and proven in the Appendix.

II.3.3 Task Execution Model

In task-based programming, applications are structured as small functional units of code called task. Each of these units will define a different task type, and during execution, many instances of each type may be spawned to operate on different pieces of the data and submitted to a runtime system for scheduling. Thus, from the runtime system’s point of view, an execution of a task-based application $P$ can be seen as a task-schedule (or a set of task-schedule, one per core), which is a sequence of $m$ task instances $S = T_1T_2\ldots T_m$.

Each task instance $T_i$ has a type associated given by $t(T_i)$. The set of all tasks instances that appear when executing $P$ will be denoted by $\mathcal{T}_P$. The number of task types of $P$ is commonly known as the task diversity of the application.

II.3.4 Equivalence Between Schedules and Memory Accesses

It is possible to show that both models (execution of an application as a sequence memory accesses vs as a sequence of tasks) presented in Sections II.3.1 and II.3.3 are equivalent for the same schedule. That is, whether we track memory reuses via tasks or as part of the application, the results are the same.

The equivalence becomes a key property for this work, as it enables the following feature: If we have the sequence of memory accesses $E$ of a task-based application, we can infer the schedule $S$, and therefore the respective execution sequence for each task $T$ in $S$, $E_T$. Conversely, if we know the task-schedule of an application and their execution sequences $E_T$, we can infer the

$^2$Note that these properties have never been formally described.
execution sequence of the entire application. This enables us to compute the sequential memory accesses for an arbitrary schedule, which allows us to use statistical modeling to analyze cache behavior for arbitrary schedules from a single profiling run.

To show this equivalence, we first show how to get a mapping from the execution of a task $T_i$ in a schedule $S$ to a sequence of memory accesses $E_{T_i} = x_1^{T_i} \ldots x_i^{T_i}$ with $l_i$ the last memory access index of task $i$, $\forall i$. This is done by defining the linear operator $\mathcal{M}$ which gives the sequence of memory accesses per task by

$$\mathcal{M}(T_i) = E_{T_i} = x_1^{T_i} \ldots x_i^{T_i} = [x_1^{T_i}, x_i^{T_i}] \forall i.$$  

Then, we use this operator to obtain the memory access sequence of an application $P$ with schedule $S = T_1 \ldots T_m$ by observing that

$$E_S = \mathcal{M}(S) = \mathcal{M}(T_1) \cdot \mathcal{M}(T_2) \ldots \mathcal{M}(T_m) = E_{T_1} E_{T_2} \ldots E_{T_m} = x_1^{T_1} \ldots x_i^{T_i} x_1^{T_2} \ldots x_i^{T_2} \ldots x_1^{T_m}$$

By doing this, we have used the operator $\mathcal{M}$ to reconstruct the execution trace $E$ from the task-schedule $S$, which shows that if we have an arbitrary task schedule, we can now reconstruct the sequential execution trace.

To complete the equivalence between both models, it is necessary to show the counterpart, where from a memory access trace, the task schedule is inferred. For this, it is necessary to know which task is executing each memory accesses. To do so, the task information is saved along with each memory access. This new definition allow us to prove the equivalence as shown in the Appendix.

II.4 Statistical Cache Modeling with Task Support

II.4.1 Existing Statistical Cache Models

In this section we present the main contribution of this paper: a formal model to predict cache behavior for different schedules from a single execution profile. This model is a solid source of insight and analysis for data-locality related bottlenecks that can leverage the performance of task-parallel applications. We will do so by first introducing the theory behind existing statistical cache models and its limitations for the task based applications, and then extend it to support task-based programs.

Figure 11 shows how existing statistical cache models represent the memory accesses (execution) of a serial application, displaying reuses between memory accesses as arcs. For instance, $(x_1, x_{k+1})$ is a reuse with reuse distance $\delta_{\mathcal{M}}(x_1, x_{k+1}) = k$.

\textsuperscript{3}For a particular input data set.
Figure 11. Sequence of Memory Accesses and reuse pairs: the same reuse pairs are shown for traditional sequential access models and our new task-based model. Note how the reuses change as the tasks are rescheduled between 12 and 13.

Figure 12. Reuse distances for tasks scheduled by \( S_{GOOD} \).

Figure 13. Reuse distances for tasks scheduled by \( S_{BAD} \).

To estimate an application’s miss ratio for different cache configurations, cache models such as StatCache (more details in [49]) first looks at the reuse distance distribution by defining a reuse distance histogram with bins \( H(i) \) mapped to the number of reuse pairs with distance \( i \). Then a linear equation is solved for the bins in \( H \) and a probability function which estimates the cache misses for a fixed size cache with random replacement policy. The result is an estimation for the application’s global miss ratio for a given cache size. The probability function is tied to the replacement policy of the cache, its size and associativity.
Implementations of these techniques are fast and have a low-overhead for data collection. Their performance is largely due to sparse data sampling and a very fast modeling step that allows the miss ratio to be calculated for any cache size quickly. Their input information is a (sampled) subset of the reuse pairs in $\mathcal{R}_E$. Since time (instruction order) is also kept for each access, the distances between accesses are straightforward to calculate, which makes it easy to build the histogram $H$.

However, existing models assume that the order of the memory accesses is the same for different sequential executions. In Figure 10, it is shown that a change in task scheduling can affect an application’s cache miss ratio. Figure 12 and 13 show the effect on the reused data from a change in schedule, because different task schedules lead to a different sequence of memory accesses.

The same memory access sequence from Figure 11 is depicted in Figure 12, but for a task-based program. Each box represents a task ($A$, $B$, $C$). In this case, the distances between the pairs from $A$ to $B$, $(x_A^1, x_B^1)$, are increased by $C$’s accesses, which may evict some of task $A$’s thereby causing task $B$ to miss in the cache if it is not large enough. This displacement is shown as $|\mathcal{M}(C)|$, which is the length of the sequence of accesses of task $C$.

Formally, each task-schedule $S$ generates a particular memory access sequence $E_S$, and therefore a different set of consecutive reuses $\mathcal{R}^{E_S}$. These define a particular schedule-dependent reuse distance operator $\delta_{\mathcal{R}^{E_S}}$. Reuse distances, and consequently their distributions $H$, are now dependent on the schedule chosen.

Existing statistical cache models do not take this into account: two different schedules will be considered as two different applications giving misleading information about the application’s data locality and preventing the analysis of schedules that have not been profiled.

### II.4.2 The StatTask Model

Figure 6 compares traditional statistical cache modeling and the StatTask model. It shows how the StatTask model can use a single profiling run to re-order the reuse pairs, generating the memory access stream that would have been seen in the execution of another schedule. From these new reuse pairs, existing statistical cache models can be used without having to re-profile the actual execution.

To estimate the miss ratio under an arbitrary schedule $S$ for a task-based program $P$, existing statistical cache models require a profiling execution such that $\mathcal{R}^{E_S}$ is captured. If we are interested in modeling a different schedule, $S'$, the same process needs to be repeated increasing the overhead significantly. StatTask allows us to evaluate arbitrary schedules $S'$ from a profile of schedule $S$ by:
1. Profiling the execution of any particular task-schedule $S$, and saving the information about consecutive reuses.
2. Calculating the reuse distances for each reuse pair.
3. Rebuilding the set of reuses that would have appeared if another schedule $S'$ would have been executed, based on the already profiled reuse pairs.
4. Calculating the reuses distances of the new reuse pairs for $S'$, which can then be used with existing statistical cache models to estimate miss ratio of different schedules and cache sizes

II.4.3 Methodology of the StatTask Model

To show how the model works, we first analyze how different types of reuses change as we change the task schedule. As mentioned in Section II.2 reuses can be private or shared. When the schedule changes, the distances of private reuses are offset relatively to the start of the task. This means that the reuse distance of private reuses is the same across all execution schedules, as the task schedule does not change the sequence of accesses within the task. An example of this is shown in Figs. 12 and 13 with $(x_{1i}^{C}, x_{l1}^{C})$. On the other hand, shared reuse distances may change with the schedule, as happens in the same figures with $(x_{1i}^{A}, x_{1i}^{B})$.

In order to classify reuses as private or shared, we will use the task property ($T$) of the memory accesses. These identifiers enable the characterization of the reuse pairs $(x_j, x_k)$. If a reuse is private, then both accesses $x_j$ and $x_k$ will be generated by the same task, thus making $T(x_j) = T(x_k)$. Otherwise it is a shared reuse. This classification is used to calculate the set of reuses of a new schedule without executing it.

Calculating the reuse pairs

This section shows how the reuse pairs are generated for a new schedule $S' = T_1 \ldots T_{m'}$. From the information we already have from profiling schedule $S$, we can make the following observations about what we expect under the schedule $S'$:

1. The private reuses of each task in $S$ will appear unmodified in $S'$, since they are relatively offset and not affected.
2. Some shared reuses between different tasks $T$ and $T'$ that appear in schedule $S$, can also appear when executing $S'$.
3. Some shared reuses between different tasks $T$ and $T'$ may appear in $S'$, but were not captured in the profile of $S$. These need to be created from the global set of reuses of the application.
Formally, StatTask defines a particular set $Q$, and it is proven that this set is actually the set of reuses that will be observed if the new schedule is executed (i.e. $Q = R^{E_{S'}}$, proof in the Appendix).

The definition of $Q$ is based on several smaller sets, as follows, where for each $T, T' \in \mathcal{T}$ we have:

$$Q_{T,T} = (T \rightarrow T)_S$$
$$Q_{T,T'} = \{ (x_k, x_j) \in \mathcal{C}(T, T') | \forall T_q \in S' : T(x_k) < T_q < T(x_j) \Rightarrow a(T_q) \notin a(T) \}$$
$$Q = \bigcup_{\forall T, T' \in S'} Q_{T,T'}.$$  

$(T \rightarrow T)_S$ denotes the set of private reuse pairs of task $T$ in schedule $S$ and $\mathcal{C}$ denotes the transitive closure operator.

**Calculating the reuse distances**

Finally, we show how to predict the reuse distances of schedule $S'$ from the distances profiled from schedule $S$. We expect the following from the distances of the reuses in $S'$: In a private reuse, its distance is the same in any schedule. However, shared reuses can either carry over to the new schedule $S'$ (kept from the profiled execution) or appear/disappear due to the new schedule via StatTask’s analysis.

If the reuse pair is carried over, computing the reuse distance in the new schedule must account (add or subtract) for the number of independent memory access corresponding to each task in-between the new schedule. On the other hand, if the reuse pair is generated transitively by StatTask, its reuse distance is computed in terms of all the distances for the pairs used to obtain the reuse.

Formally, StatTask defines a function $\gamma$ over the reuses for the new schedule $S'$, and we prove that this function is in fact the reuse distance function that would be observed if $S'$ is executed (i.e. $\gamma = \delta_{R^{E_{S'}}}$, proof in the Appendix).

The function $\gamma$ is given by

$$(x_j, x_k) \in (T \rightarrow T)_{S'} \Rightarrow \gamma(x_j, x_k) = \delta_{R^{E_{S}}}(x_j, x_k).$$
$$(x_j, x_k) \in (T \rightarrow T')_{S'} \Rightarrow \gamma(x_j, x_k) = \delta_{R^{E_{S}}}(x_j, x_k) - \nu_{S}(T, T') + \nu_{S'}(T, T').$$

The operator $\nu_S$, when applied to two tasks, will give the number of accesses of the tasks exactly between $T$ and $T'$ in schedule $S$ ($\delta_R$ over a pair $(x_j, x_k)$ gives the distance within the set of global reuses).

StatTask opens the door to a new category of tools and predictive analyses, as metrics such as cache miss ratios for arbitrary cache sizes can be obtained for arbitrary task schedules, without additional profiling and execution.
II.5 Evaluation

To evaluate StatTask, we implemented it using the Pin[87] binary instrumentation tool to profile applications, and compared StatTask’s ability to predict cache miss ratios for different schedules across the BOTS benchmark suite. Our tool generates a profile by sampling 1/1000 of the memory accesses (randomly selected, exponentially distributed) and collecting the corresponding address, reuse distance, task id, program counter, and instruction count. Our sampling approach follows the one described in [49], yielding a 20% sampling overhead. By using Pin for dynamic binary instrumentation we avoid the need to modify the applications’ source code, being transparent to the programmer and runtime, and enabling the same methodology to be applied in other task-based frameworks such as StarPU, Intel’s Cilk, Intel’s TBB and OmpSs. Despite the data collection is done in a single threaded execution, StatTask allows to model multi-core executions based on the per-task reuses. The resulting profile is consistent with both formalisms presented in Sections II.3.1 and II.3.3.

After profiling, the StatTask model is run with the desired schedule and cache size as inputs. StatTask rebuilds the reuse pairs and their reuse distances according to the desired schedule, and uses this information to calculate the cache miss ratio for the given cache size with the StatStack[57] statistical model. Reference results are obtained by measuring the L3 miss ratio via hardware performance counters for a single-threaded execution of the specific schedule on a machine with the modeled cache size. To avoid calculating expensive transitive closures (Section II.4.3), we implemented reuse unfolding, wherein the reuses are annotated with starting and ending accesses in a time-ordered table for rapid analysis.

Figure 14 shows StatTask’s accuracy across the BOTS benchmarks for different schedules and input sizes. Each benchmark was sampled when run with multiple different schedules (S1, S2, S3) and we present results for StatTask’s prediction for the same schedules compared to measured results for each schedule.

StatTask’s modeled cache miss ratios (dark green bars) show an 8.7% average error (max 19%) with respect to the measured cache miss ratios (light green bars). The inaccuracies are a result of our 1-in-1000 memory access sampling rate, and can be reduced by increasing the sampling rate at a cost of increased overhead. These results demonstrate that StatTask is able to accurately predict the cache miss ratio for a variety of schedules.

To evaluate StatTask’s ability to model arbitrary schedules from a single input, we gathered profile data for schedules S1 and S2 and used StatTask to predict for all possible combinations: S1→S1, S1→S2, S2→S1, S2→S2. Figure 15 shows StatTask’s accuracy across these different schedule/modeling combinations, and demonstrates that StatTask can accurately predict the miss ratio for schedules other than the one used in sampling.
To evaluate StatTask’s sensitivity to changes in input data, we compared several of the benchmarks (sparselu, strassen, sort, floorplan, nqueens) with varying inputs: three different small and large datasets (blue bars) across multiple schedules\(^1\). Figure 16 shows that several of the benchmarks see little change in behavior with varying input data, as the per-task execution and the overall distribution of tasks varies little with the input data. For these applications StatTask can accurately predict schedule behavior even in the presence of data set changes.

However, several applications show large changes in cache miss behavior as the size of their data sets change. Figure 14 highlights how the behavior of floorplan and nqueens changes as their data set sizes are increased by 4× and 3×, respectively, resulting in 40× and 120× increases in execution time. The resulting change in cache behavior is due to the increase in locality

\(^1\)The sizes of the inputs were all within 5%.
from larger problem sizes, as they spend more time working on each part of the data, resulting in more cache hits. StatTask could be adapted to handle such drastic changes in behavior by incorporating a dynamic profiling phase and extrapolating the modeled data to fit dynamically profiled data.

Our implementation and evaluation of StatTask across the BOTS benchmarks demonstrates that this technique is both capable accurately predicting the cache behavior of task-based programs and flexible enough to predict the behavior of different schedules from the originally profiled one.

II.6 Related Work

There are three categories of related work: existing profiling tools that identify bottlenecks of task-based applications, task-scheduling optimization techniques, and finally techniques to analyze and understand data locality properties of applications.

Many tools exist to profile scheduling and load-balancing of tasks. Ding et. al. [54] presented a generic and accessible tool for task monitoring, independent of any program or library and able to acquire rich information with very low overhead, targeting load balancing and scheduling problems unrelated to data reuse. Lorenz et. al developed [60], a library for identifying performance problems inherent to tasking with OpenMP through direct in-

Figure 15. Using StatTask to model different schedules from the same profiled data. (e.g., S1→S1, S1→S2, S2→S1, S2→S2)
Figure 16. Sensitivity to different input data sets of the same size.

None of these approaches for task-based profiling have incorporated a general method for understanding the data reuse implications of the tasks and schedules. In this category, characterization of data reuse has been done theoretically in [45] by Frigo. Practically, this can be done through instrumentation based techniques as presented by Aamer et. al. in [59] and Weidendorfer in [64].

Statistical cache modeling, first introduced in [49], is another widely used way to characterize data locality. This work has been extended to other cache replacement policies by Eklov in [57], and to support thread-based or multi-core shared caches in [48, 56].

II.7 Conclusion and Future Work

This work addresses the interplay between task scheduling and shared data reuse through caches. We have presented StatTask, a new statistical model...
that can sample a single, serial run of a task-based application build reuse graphs, reuse sets, and distance metrics. From these, StatTask can accurately predict cache behavior for arbitrary schedules of the tasks and cache sizes on multicore parallel systems. We have implemented a tool, which requires no programmer annotation, code or runtime changes, analyzed a range of benchmarks and shown that there is a potential to share an average of 35% of the memory accesses between tasks (up to 80%), while demonstrated how this analysis can be used to better understand the sharing characteristics.

With StatTask we have a new ability to rapidly explore the impact of task scheduling on cache behavior, which opens up a range of possibilities for intelligent, reuse-aware schedulers and better performance.

We see several ways to move forward, the main one being to incorporate this technique within a task runtime system to profile the applications and tasks being executed, create profiles, and use this information during future runs to identify reuse patterns, predict miss ratios and modify task placement.

Regarding tools to analyze data reuse and provide insight to developers, several optimizations can be done to achieve better analysis modeling performance, as well as more intelligent sampling by, for instance, phase-guided sampling to avoid collecting redundant information for private reuses while getting more dense samples on shared ones.

II.8 Appendix: Proofs

Lemma. Let $E$ and $E'$ be execution traces such that they share the exact same set of accesses, but in different order. Then

$$(x_j, x_k) \in \tilde{R}^E \Rightarrow (x_j, x_k) \in \tilde{R}^{E'} \vee (x_k, x_j) \in \tilde{R}^{E'}.$$  

Proof. Let $(x_j, x_k)$ be an element of $\tilde{R}^E$. By definition, $a^j = a^k$. Since $E$ and $E'$ share the same set of accesses, there exist $x_0$ and $x_1$ in $E'$ such that $x_j = x_0$ and $x_k = x_1$. Lets assume $x_0 < x_1$, since $a^j = a^k$ then $(x_0, x_1) \in \tilde{R}^{E'}$. If $x_1 < x_0$ then $(x_0, x_1) \in \tilde{R}^{E'}$.  

Lemma. $M^{-1}$ and $M$ are inverses.

Proof. Let $T$ be a task, such that $E_T = x_1 \ldots x_r$. We can see that

$$M(M^{-1}(x_1 \ldots x_r)) = M(T(x_1)) = M(T) = E_T = x_1 \ldots x_r$$

Conversely,

$$M^{-1}(M(T)) = M^{-1}(E_T) = M^{-1}(x_1 \ldots x_r) = T(x_1) = T$$

Theorem. $Q = \tilde{R}^{E \S}$
Proof. It is straightforward to prove that \( Q \subseteq R^{E_S'} \). It is enough to observe that \( \mathcal{C}_S(T, T') \) gives all the pairs in \( R^{E_S} \) that start in \( T \) and end in \( T' \). Since this set is execution independent, those pairs are also in \( R^{E_S'} \). The condition \( \forall T_q \in S' \) such that \( T(x_k) < T_q < T(x_j) \Rightarrow a^j \notin a(T_q) \) filters out the non-consecutive reuses. Therefore, all the elements of \( Q \) are consecutive reuses.

We will now show an outline for the proof that \( R^{E_S'} \subseteq Q \). If \( (x_j, x_{j+d}) \in (T_0 \rightarrow T_0)_{S'} \), then \( T(x_j) = T(x_{j+d}) = T_0 \). As \( S \) and \( S' \) use the same task universe, \( \exists T_r \in S \) such that \( T_0^S = T_r^{S'} \). By Lemma 1, \( E_{T_0} = E_{T_r} \). Therefore, \( \exists x_p \in E_{T_r} \) such that \( x_p = x_j \) and \( x_p + d = x_{j+d} \), as private reuses are relatively offset. Then \( (x_j, x_{j+d}) \in (T_r \rightarrow T_r)_S \subseteq Q_{T_r, T_r} \subseteq Q \).

Let's now consider the case \( (x_j, x_{j+d}) \in (T_n \rightarrow T_m)_{S'} \). The tasks \( T_n \) and \( T_m \) also occur in \( S \). We will assume that \( T_n < T_m \). Let \( T_1, \ldots, T_k \) such that \( T_n < T_1 < \cdots < T_k < T_m \). The proof is by induction on \( k \).

When \( k = 0 \), then the sequence \( T_n T_m \) occur in \( S \). Therefore, it exists \( x_r \in E_S \) such \( [x_j, x_{j+d}] = [x_r, x_{r+d}] \). Since \( x_r = x_j \) and \( (x_j, x_{j+d}) \in R^{E_S'} \), we know that \( \forall x_r < x_s < x_{r+d}, d^r \neq d^s \). Therefore \( (x_j, x_{j+d}) = (x_r, x_{r+d}) \in (T_n \rightarrow T_m)_0 \subseteq Q_{T_n, T_m} \subseteq Q \).

When \( k = 1 \), then \( T_n T_{n+1} T_m \in S \). Let's assume that \( a(T_{n+1}) \neq a(T_n) \), thus \( \forall x_s \) such that \( x_j < x_s < x_{j+d} \Rightarrow a^s \neq a^j \). Therefore \( (x_j, x_{j+d}) \in (T_n \rightarrow T_m) \subseteq Q_{T_n, T_m} \).

If that does not happen, it is enough to assume that there are unique \( x_1, \ldots, x_d \) such that \( x_j < x_1 < \cdots < x_q < x_{j+d} \) and that \( a^j = a^1 = \cdots = a^d = a^{j+d} \), with all the accesses in between with different addresses. This means that the pairs \( (x_j, x_1), (x_1, x_2), \ldots, (x_q, x_{j+d}) \) are elements of \( R^{E_S} \). Therefore, by definition of \( \mathcal{C} \), this means that \( (x_j, x_{j+d}) \in \mathcal{C}(T_n, T_m) \subseteq Q_{T_n, T_m} \subseteq Q \).

The final case is \( k \Rightarrow k+1 \). Let \( T_n T_1 \ldots T_{k+1} T_m \in S \). Two cases are necessary to prove. The first one, where the set of addresses are of tasks \( T_1, \ldots, T_k \) are disjoint from \( T_n, T_m \). If that happens, then the only thing left to check is the set of addresses of \( T_k \), which is analogous to the case \( k = 1 \) for \( T_{k+1} \). Otherwise, a unique number of accesses with the same address appear in \( T_1, \ldots, T_k \), which by inductive hypothesis can be used transitively to obtain a pair in \( Q \).

The proof for when \( T_m < T_n \) is analogous.

\[ \gamma = \delta_{R^{E_S'}} \]

Proof. Let \( (x_j, x_k) \in R^{E_S'} \), and let \( T(x_j) = T(x_k) \). Let \( T_{n_1}, \ldots, T_{n_s} \) be such that \( T_{j_1} T_{n_1} \ldots T_{n_s} T_{k} \in S' \). These are the tasks scheduled between the starting and ending tasks causing the reuse in \( S' \). Let's also assume \( T_{m_1}, \ldots, T_{m_s} \) such that \( T_{j_1} T_{m_1} \ldots T_{m_s} T_{k} \in S \), thus representing the tasks between the starting and ending tasks of the reuse in \( S \). It is easy to see the following memory access sequence when \( S' \) is executed:

\[ x_j \ldots x_l x_1 \ldots x_l \ldots x_l \ldots x_l x_1 \ldots x_k = x_j \ldots x_l T_{j_1} E_{T_{n_1}} \ldots E_{T_{n_s}} x_1 \ldots x_k \]
Therefore, since the access distance is linear, we can see that
\[
\delta_{\mathcal{R}E_S}(x_j, x_k) = \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^j}}) + |E_{T_{n_1}}| + \cdots + |E_{T_{n_r}}| + \delta_{\mathcal{R}E_{S'}}(x_{1_{T_x^k}}, x_k)
\]
\[
= \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^j}}) + |\mathcal{M}(T_{n_1})| + \cdots + |\mathcal{M}(T_{n_r})| + \delta_{\mathcal{R}E_{S'}}(x_{1_{T_x^k}}, x_k)
\]
\[
= \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^j}}) + \gamma_S(T_{x_j}, T_{x_k}) + \delta_{\mathcal{R}E_{S'}}(x_{1_{T_x^k}}, x_k)
\]

On the other hand, we can also see the following sequence when \(S\) is executed:
\[
x_j \ldots x_{L_{T_x^j}} x_{1_{T_x^1}} \ldots x_{1_{T_x^{m_1}}} \ldots x_{1_{T_x^{m_s}}} x_{1_{T_x^k}} \ldots x_k,
\]

analogously, see that \(\delta_{\mathcal{R}E_S}(x_j, x_k) = \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^j}}) + \gamma_S(T_{x_j}, T_{x_k}) + \delta_{\mathcal{R}E_{S'}}(x_{1_{T_x^k}}, x_k),\n\]

and therefore, \(\delta_{\mathcal{R}E_S}(x_j, x_{1_{T_x^1}}) + \delta_{\mathcal{R}E_S}(x_{1_{T_x^k}}, x_k) = \delta_{\mathcal{R}E_S}(x_j, x_k) - \gamma_S(T_{x_j}, T_{x_k}).\n\]

Since the sequence \(x_j \ldots x_{L_{T_x^j}}\) is identical both in \(E_S\) and \(E_{S'}\) then
\[
\delta_{\mathcal{R}E_S}(x_j, x_{1_{T_x^1}}) = \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^1}}).\n\]

The same holds for \(x_{1_{T_x^k}} \ldots x_k.\) Then,
\[
\delta_{\mathcal{R}E_S}(x_j, x_k) = \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^1}}) + \gamma_{S'}(T_{x_j}, T_{x_k}) + \delta_{\mathcal{R}E_{S'}}(x_{1_{T_x^k}}, x_k)
\]
\[
= \delta_{\mathcal{R}E_{S'}}(x_j, x_{1_{T_x^1}}) + \delta_{\mathcal{R}E_{S'}}(x_{1_{T_x^k}}, x_k) + \gamma_{S'}(T_{x_j}, T_{x_k})
\]
\[
= \delta_{\mathcal{R}E_S}(x_j, x_k) - \gamma_S(T_{x_j}, T_{x_k}) + \gamma_{S'}(T_{x_j}, T_{x_k})
\]
\[
= \gamma(x_j, x_k)
\]
References


Paper III

TaskInsight

Understanding Task Schedules Effects on Memory and Performance

Germán Ceballos, Thomas Grass, Andra Hugo and David Black-Schaffer

In Proceedings of the
International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)
Austin, Texas, USA, February 2017
Abstract
Recent scheduling heuristics for task-based applications have managed to improve their performance by taking into account memory-related properties such as data locality and cache sharing. However, there is still a general lack of tools that can provide insights into why, and where, different schedulers improve memory behavior, and how this is related to the applications’ performance.

To address this, we present TaskInsight, a technique to characterize the memory behavior of different task schedulers through the analysis of data reuse between tasks. TaskInsight provides high-level, quantitative information that can be correlated with tasks’ performance variation over time to understand data reuse through the caches due to scheduling choices. TaskInsight is useful to diagnose and identify which scheduling decisions affected performance, when they were taken, and why the performance changed, both in single and multi-threaded executions.

We demonstrate how TaskInsight can diagnose examples where poor scheduling caused over 10% difference in performance for tasks of the same type, due to changes in the tasks’ data reuse through the private and shared caches, in single and multi-threaded executions of the same application. This flexible insight is key for optimization in many contexts, including data locality, throughput, memory footprint or even energy efficiency.

III.1 Introduction
Scheduling task-based applications has become significantly more difficult due to the growing complexity of computer architectures. Typical approaches for optimizing scheduling algorithms consist of either providing an interactive visualization of the execution trace [85, 80] or simulating the tasks execution to evaluate the overall scheduling policy in a controlled environment [92, 83]. The developer then has to analyze the resulting profiling information and deduce if the scheduler behaves as expected, and qualitatively compare different schedulers.

Poor scheduling decisions often cause performance variations across tasks of the same type, which makes it hard to identify the root cause from the overall schedule. Existent work [93] proposed scheduling strategies that include these performance differences in the load-balancing algorithm to overcome this problem. 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.

The effects of poor scheduling decisions can be most easily seen in idle execution time due to load imbalance from the inability to prioritize tasks on the critical path or appropriately map tasks to processors. However, scheduler decisions also impact data locality in the cache hierarchy by changing the order of producer and consumer tasks. The result of these decisions is performance variation across tasks of the same type, which can only be understood by analyzing how the tasks share data and how the schedule affects that sharing.

Generally, task-based application developers blame this performance degradation on data locality and attempt to characterize their workload based on data reuse without considering the dynamic interaction between the sched-
This is simply because there has been no way to obtain precise information on how the data was reused through the execution of an application, such as how long it remained in the caches, and how the scheduling decisions influenced the reuse history. Without an automatic tool capable of providing insight as to whether and where the scheduler misbehaved, the programmer must rely primarily on intuition to understand and adjust the scheduler for improved performance.

In this paper, we present TaskInsight, 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 through the cache hierarchy. We show how the reuse of data throughout the execution can provide insights into the performance of the scheduler, regardless if it is optimized for data locality, bandwidth, memory footprint, etc. Further, TaskInsight can interface directly with the task-based runtime system to provide this information both to the programmer and the scheduler.

Previous work [89] has shown the effects of data reuse distances in performance degradation. Those results were based on aggregated statistics and do not provide the necessary detail to manually (developer) or automatically (runtime system) adjust the schedule to improve performance or locality. Scheduler optimization is a notoriously difficult problem as past decisions affect choices and performance in the future, making it hard to explain performance without a detailed view across the program.

In order to understand the performance of a particular schedule, and thereby the scheduler itself, it is therefore necessary to address three critical questions:

- (Q1) **What** scheduling decisions influenced the performance of the execution?
- (Q2) **When** did those decisions happen?
- (Q3) **Why** did those decisions affect the performance?

Answering these questions is vital for dynamic scheduling strategies that adjust their decisions in real time based on how tasks use the hardware resources. Scheduling decisions need to take into account the individual task
performance to optimize the overall application, which is nearly impossible without answers to the above questions.

In this paper we introduce the TaskInsight methodology, which shows how data reuses between tasks can provide key information for answering these questions, as they can be quantified in time, and thereby expose the interactions between the tasks’ performance and their schedule. We make the following contributions:

1. A novel classification of the data of each task based on when the data is used over time. This classification is able to expose different memory behaviors inherent to the schedule.

2. A new analysis of schedulers based on the preservation of temporal locality of the data through time, by connecting our classification to the measured performance results and statistics from the private caches.

3. A new technique to analyze schedulers based on the preservation of spatial locality of the data through time, by linking our classification with performance results and statistics from the shared caches.

We start with an example that shows how the overall performance of an application changes when executing with different schedules due to an increase in last-level cache misses (Section III.2). We then propose a profiling tool and TaskInsight’s data classification technique that allows to clearly differentiate the schedules in terms of their data reuse patterns, using a data reuse graph as in [81] (Section III.3). Later, we show how to connect this classification to changes in data reuse, changes in cache misses and changes in performance during the execution: first from the perspective of the private caches (temporal locality on a single-threaded execution, Section III.4) and later from the shared caches (spatial locality on multi-threaded run, Section III.5).

III.2 Motivation

It is well known that cache optimization is crucial for performance, and we begin by illustrating these effects on a task-based setting to understand the main driver behind TaskInsight.

We consider a simple example in a simulated environment that allows us to precisely control the effects of memory bottlenecks: a task-based implementation of the Cholesky Factorization using the OmpSs runtime [86]. The input is a 32MB matrix with 256x256 block size, which is enough to hold one task’s dataset at a time in 2MB last level cache. The application generates a total of 120 tasks of four different types (gemm, potrf, syrk, and trsm). We study the performance over time in a simulated single-threaded execution using the TaskSim simulator [91, 90].

2For the case study, we chose the default configuration, using a 2MB last level shared cache. Section III.5 shows results for native runs on real hardware.
Figure 31 shows the total cycle count (execution time of the tasks), total number of last level cache misses and average task last level cache misses-per-kilo-instruction (mpki), for two different executed schedules, provided by the OmpSs runtime. The first policy, naive, uses a breadth-first-search policy, scheduling tasks in creation order, while smart schedules tasks according to a heuristic, wherein child tasks are prioritized over the next task in the breadth-first order. This heuristic optimizes for locality, as child tasks are more likely to reuse data from their parent.

Note that we are reporting performance in terms of task cycle count. This metric counts just the time it takes to execute the tasks, and ignores overheads for the scheduler, runtime system, and load imbalance. By measuring just the task cycle count, we can accurately capture changes in the performance of specific tasks, which is essential to understand the impact of the scheduler on task performance.

As we see from the total cycle count, smart is 6% faster than the naive scheduler. From the cycle count breakdown we see that the main difference comes from the number of cycles spent on DRAM accesses as a result of a 5% increase in last-level cache misses. This results in a 14% increase in the average task MPKI. However, while these statistics clearly show that the memory behavior and performance are affected by the scheduling choice, the overall statistics are not detailed enough to provide actionable insight into the reasons for the increase in cache misses, when they happen, or what potential there is for improvement.

Existing tools follow a similar approach and propose analyzing the performance of the overall application [83] or of the different tasks independently [85]. However, they lack information on how the tasks influence one another through the execution (schedule and caches) and how the algorithmic decisions of the scheduler can impact the performance of the tasks and thus of the application. Such information is necessary not only for evaluating performance, but also for improving the scheduler.

TaskInsight covers this missing area by inferring quantitative metrics about the scheduler’s impact on data reuse between tasks and connecting it to performance analysis of different schedules.

In the next section, we show how we can combine the execution schedule with a new data classification to build a data reuse graph, which exposes both potential and actual data reuse in a hardware-independent manner. We then analyze the behavior of the schedule under specific hardware configurations, such as cache sizes, to determine the schedule’s impact on a given system. This flexibility allows us to explain not only the impact of different schedules on different systems, but to also work backwards to understand how earlier scheduling decisions affected the performance of later tasks due to data reuse through the caches.

---

3The reader can click on the figures to see an online interactive version of the data.
III.3 Through the data-reuse glass

Typically each task in an application operates on its own data set, but, over time, parts of a task’s data may be reused by later tasks. This means that a portion of the data set can be considered shared. If the scheduler can arrange to execute the tasks close enough together in time, it will increase the chance that the shared data is in the cache, and thereby improve performance through temporal locality.

To understand the impact of these scheduling decisions, it is first necessary to analyze how much shared and private data each task has. TaskInsight does this by profiling the execution to sample the memory addresses for each task. Once the profile is collected, TaskInsight makes 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 used by the immediately previous task before the current one.
- **2nd-last-reuse**: the memory address was used by the second-to-last task before the current one.
- **older-reuse**: the address was used by a task that came more than two tasks previous to the current one.

Figure 32 compares the cumulative amount of data touched as the program executes between the two schedules from Figure 31. 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 Figure 32 we can see how the naïve schedule (red curve) executes tasks in a way that touches new data much more aggressively, in bursts. On the other hand, the smart schedule (orange curve) is much smoother, meaning that new data is brought at a slower rate. The flat regions on the curves indicate reuse-periods, where the scheduled tasks operate on previously used data, and therefore do 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 clearly result in different cache miss rates, and therefore impact performance when executed.

Although the new-data category intuitively exposes the rate at which the applications install new data in the caches, it does not explain how the shared data is 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 33 and 34 show the breakdown of memory accesses (new-data, last-reuse, 2nd-last-reuse, older-reuse) for both schedules (naïve, smart) 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 lower-middle region shows the percentage of second-last memory accesses, and finally the upper-middle 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 policy, compared to the naïve approach, which touches most new-data during the first 16 tasks. In addition, the area corresponding to last-reuses increases considerably (more dark blue area) in the smart schedule, meaning that more shared data is being reused sooner. This is also observed between tasks 100 to 115: in the naïve schedule, most of the data used is coming from the second-last predecessor, but in the smart schedule, data is coming from its immediate predecessor. As the immediate predecessor is more recent, one would intuitively expect that this schedule would result in a higher cache hit rate and better performance.

With this classification, it is now clear that the two schedules have very different reuse characteristics. However, we need to translate the observations from the previous figures into relevant metrics to compare the schedules overall. TaskInsight uses aggregated statistics from each reuse category over time to understand how reuses flow from one category to another.

Figure 35 shows an example of this. It displays the percentage of memory accesses corresponding to each category (y-axis, %), as a function of time (x-axis; task number), for both the naïve (left) and smart (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 scheduler has 11% more last-reuses than naïve. 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 schedule turning
Figure 19. Data reuse for na"ive schedule: new data is brought in big chunks, and data is reused far away, exposing more older-reuses during the execution.

Figure 20. Data reuse for smart schedule: new data is brought more sparsely in smaller chunks, and data is reused sooner showing more last-reuses. 2nd-last-reuses into last-reuses, while the remaining 6% comes from improving older reuses.
III.4 Analyzing Performance

The classification described in the previous section allows us to characterize the impact of different schedules on memory behavior in a hardware-agnostic manner. However, comparing the relative differences between these metrics is not enough to predict how they will affect performance. To accomplish this, TaskInsight combines the data reuse classification with performance measurements to explain changes in performance due to changes in data access.

Figure 21. Breakdown of reuses per category: naive (left) vs smart (right).

Figure 35 not only allows us to see what aspects of the schedules are different, but also to precisely detect when the schedules have differences in data reuse. Since the tasks are uniquely identified, it is possible to point to the specific tasks that benefited from the rescheduling or were hurt by it. In the following section, we show how to connect this classification with performance measurements to also understand why performance was affected by changes in memory behavior due to scheduling.

Overall, the TaskInsight analysis allows us to understand the impact of scheduling changes in a way that can be used to improve performance by increasing reuse through caches. Approaches that only measure the actual cache miss ratios per task (e.g., 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 novel methodology enables scheduler designers to gain insight into how specific scheduling decisions impact later tasks.
We will first use TaskInsight to study how scheduling affects performance due to changes in the data’s temporal locality, and how it is related to cache reuse. To illustrate the analysis, we consider the simulated single-threaded execution of the Cholesky factorization from Section III.2, which exposed a performance difference between two schedules. Focusing on single threaded execution allows us to exclude contention for the shared last-level cache. Section III.5 later extends TaskInsight to analyze multi-threaded runs, both regarding temporal and spatial locality changes due to scheduling, and explaining performance changes.

During the Cholesky factorization execution, we measure the performance of each task (average cycles-per-instruction, CPI) and cache behavior such as last-level cache misses and accesses. The CPI for each task instance (color-coded by type) is shown in the bottom of Figure 35. We can see that the performance across tasks (CPIs) vary far more in the naive schedule (average 20% worse). For instance, if we look at Task with ID 57, in naive it was scheduled as the 57th task, executing at 0.33 CPI. In smart, this task was executed 32nd, delivering a 15% increase in performance at 0.29 CPI.

As TaskInsight can tell exactly where the data is coming from, by correlating CPI with the data reuse classification, it is possible to see that in the first schedule the task is reusing 98.1% of its data from older tasks (highlighted in the figure). Scheduling this task sooner results in 96.5% of the data is coming from the previously executed task, increasing the likelihood that the data is reused through the cache, and thereby increasing performance.

Moreover, we also see variations in task performance within the same schedule for tasks of the same type: e.g. in smart, tasks 113-118 are all syrk with same input size, however, task 113 has a 7% worse CPI than its subsequent ones. Again, by correlating with the new-data graph, we see that this is because it is bringing 20% new data for that schedule.

In addition, 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 (size), as the amount of reuse that can be realized through the cache depends on the size of the tasks’ data.

III.5 Multi-threaded Executions

The previous section showed how TaskInsight can quantitatively characterize different schedules with regards to their temporal data locality and data
Figure 22. Performance-L2 correlation. Tasks from the same type are clustered better in the smart schedule, meaning less performance variation within the same type due to memory.

Figure 23. Performance-L3 correlation. Tasks and types exposing drastic performance degradation due to sensitivity to the shared cache. The smart schedule shows tasks with 50% less misses and 50% less performance variation within the same type.

Figure 24. Performance-Memory Correlation.

reuse through the caches for single-threaded applications. However, and unlike previous methods such as looking at aggregated hardware performance counters, our technique enables us to identify which specific scheduling decisions caused changes in memory behavior, when they occurred, and why they lead to performance loss.

When running multi-threaded applications, the complexity of the analysis is significantly increased by having multiple per core schedules executing in
parallel and the effects of shared caches, which can cause the schedules to interfere with each other. The shared cache effects can be particularly important for performance as the cost of an L3 miss is much higher than an L2 miss (and L3 hit).

When using TaskInsight for multi-threaded executions, it provides information about the changes in temporal locality on a per core basis. However, to handle parallel executions of multiple schedules it is also necessary to understand the effects in performance of the shared cache.

As multi-threaded executions can execute tasks at a far higher rate, it is often difficult to identify tasks whose performance was affected by the schedule. We begin by showing how to use performance-to-memory correlation as a filtering technique (Section III.5.1). We then explain how to use TaskInsight on a per-core basis to reveal where schedules fail to preserve temporal data locality through the private caches (Section III.5.2). Finally, we show how to extend TaskInsight to explain the effects of shared caches in performance variation, by modeling also spatial data locality of each schedule (Section III.5.3).

III.5.1 Performance-memory filtering

For our analysis of parallel tasks we now consider a larger execution of the same Cholesky factorization on a quad-core machine\(^4\). The execution consists of 796 task instances in total, working with a 256x256 block size, where 550 are \texttt{dgemm} (80% of the total cycle count), 114 are \texttt{dsyrk}, 117 are \texttt{dtrsm} and 15 are \texttt{dpotrf}. Each of these task routines are from the Intel MKL library.

We executed the application with the same two scheduling policies as before (\texttt{naive} and \texttt{smart}) on 4 cores\(^5\), and collected data from hardware performance counters including number of instructions, cycles, and L2/L3 cache misses/accesses, \textit{per-task} (implementation details in Section III.6).

When aggregating these results, we see that the total cycles speedup of \texttt{smart} against \texttt{naive} is over 10%, and when looking at the cycle breakdown, \texttt{naive} incurs 9% more L2 misses on average per core, and 40% more L3 misses than \texttt{smart}. This is a significant variation due to scheduling.

TaskInsight can determine which specific tasks were affected by the schedule differences and at which specific moments in time. However, displaying data from a large multi-threaded execution would require very complex graphs for each execution core. To see the effect of scheduling and task type on memory related performance, we start by correlating performance to memory on a per task basis in Figures 36 and 37. This presentation allows to identify which task instances had the most performance variation between the different schedules. These figures show scatter plots of per-task CPIs vs. (private) L2 miss

\(^4\)Intel Core i5-3550 CPU (Ivy Bridge). 8-way associative L1 cache, 256kB 8-way associative L2 cache, 6MB 12-way associative L3 cache, 16GB RAM.
\(^5\)One thread was pinned per core, and HyperThreading was disabled.
ratios and (shared) L3 miss ratios, respectively, colored by task type for both schedules.

The performance-to-L2 miss ratio data (Figure 36) shows which tasks are sensitive to temporal locality changes through the L2 (sensitivities to the shared L3 are discussed in Section III.5.3). As we can see, the variance in the performance of the tasks using the naïve schedule (Figure 36 left) is significantly higher, especially for task types such as dgemm (blue) and dsyrk (red). When we compare this to the smart schedule (Figure 36 right), we see that dgemm (blue), which represents 80% of the total execution time, have over 11% better CPI as a result of an 8% lower L2 miss ratio. This is seen in how the dgemm cluster moves down (lower CPI, better performance) and to the left (lower L2 miss ratio) between the naïve (left) and smart (right) plots in Figure 36. A similar effect of the smart scheduler can be seen for the dsyrk (red) task cluster.

The opposite effect can be seen, though, for the dtrsm tasks (light-green). In the smart case, the cluster shows 4% more L2 misses, causing performance to be 6% worse. Despite this, the overall performance is still better for this schedule as the dtrsm tasks represent only 8% of the total execution time. In this case the scheduler has effectively traded off worse temporal locality in the less frequent dtrsm for better locality in the far more frequent dgemm tasks.

From this correlation we can see both which task types are the most sensitive to memory effects due to scheduling, and also which tasks instances’ performance is most likely changed due to L2 caching effects and independent of caching effects. By using this filtering technique to identify these tasks, we can further study them with TaskInsight’s technique from Section III.4 to fully understand if the changes in performance are due to reuses from L2, as we show in the following section.

III.5.2 Temporal Locality of Private Caches

As each task is uniquely identified, it is possible to compare the difference of CPI and L2 miss ratio values between the two schedules on a per-task basis. This correlation allows us to distinguish between task instances without performance variation, task instances with performance variation due to non-memory effects (different CPI but no changes in L2/L3 miss ratios) and task instances with performance variation due to memory effects (both different in CPI and miss ratio). By filtering the analyzed tasks to the cases where performance variation is correlated with memory effects, we can target TaskInsight to study only on those tasks where we can benefit from scheduling insight as to how the schedule affected data reuse through the caches.
To analyze private caches, we can use the analysis from Section III.4 on a per-core basis, as the private cache behavior is only affected by the local core’s schedule. This enables the characterization of the two schedules in terms of how well they preserve temporal locality through the private caches. By only looking at the filtered tasks from the previous section, TaskInsight is able to show why those particular task instances missed more in the private L2 cache and when, which is essential to improve scheduling decisions.

### III.5.3 Locality of Shared Caches

To understand how the shared cache affects the performance of different schedules, we extend TaskInsight to provide information regarding how tasks share the last level cache, and how this affects performance across different schedulers. The novelty of this approach lies in correlating caching, changes in schedule and changes in performance. This contribution is key to characterize not only how good a schedule is in temporal locality preservation through private caches, but also through shared caches, as well as characterize them in terms of spatial locality.

The correlations between task performance (CPI) and shared cache behavior (L3 Miss Ratio) is shown in Figure 37, colored by task type. For all task types, there is a rather linear correlation, meaning that the fewer L3 cache misses, the better the performance. Furthermore, we see that almost half of the dgemm tasks (blue) experience roughly twice the performance (lower CPI) for the smart schedule (right) over the naïve schedule (left) with L3 miss ratio decreases of up to 50%.

Tasks that have a radical change in L3 misses are likely to have a performance loss, and vice-versa, so we consider those as candidates to study with TaskInsight.

While this performance-based analysis can identify which tasks saw worse performance due increases in L3 misses, it cannot give any insight into which scheduling decisions caused them. For example, in the naïve schedule, two tasks with data reuse between them may have been scheduled further apart from each other, resulting in their data being evicted from the L3 before it could be reused. Alternatively, the tasks may have been executed very close to each other, but co-executed with other tasks that used a large disjoint dataset, thereby polluting the shared cache and evicting the data before it could be shared.

TaskInsight is able to explain where the difference in L3 cache misses is coming from, by modeling each set of co-running tasks, and then computing their predecessor’s reuse (temporal reuse) and amount of data reused at the

---

6We are ignoring evictions due to the inclusive shared L3 cache as its size is far larger than the private L2s.
Determining co-running sets: for each core, the set of co-running (overlapping) tasks is identified and used to build a single set with their memory addresses during analysis.

The same time (spatial reuse). This now allows us to determine which scheduling decisions cause the increase in miss ratios, which lead to worse performance.

To undertake this analysis, we first look at the sequence of tasks executed on each core, i.e. the per-core schedules. For each task instance, we determine the set of overlapping tasks for the schedule, which is the set of tasks executing on other cores at the same time. Figure 38 shows an example of this. In this case, the first co-running set comprises tasks 7, 45, 12 and 23, which are executed in parallel from core 0’s perspective. The second set contains tasks 8, 30, 13 and 31, and so on. This analysis gives us a sequence of co-running tasks over time, across all different cores. Each core will have a different sequence, but our analysis generates similar results for each of them.

At the same time, TaskInsight builds the application’s (schedule-independent) reuse graph introduced in [81], and combines it with the co-running task sequence to compute the set of memory addresses used by each co-running set. This allows to model the sequence of co-running tasks over time, and use the analysis in Section III.4 to analyze how much data was reused over time, but in the shared cache.

The result of this analysis is shown in Figure 39: the top half of the graph corresponds to the naïve schedule and the bottom half to the smart schedule. The x-axis represents execution order of the 216 sets of co-running tasks. The top of each graph shows the classification of the reuses provided by TaskInsight as in Figure 34.
Figure 26. Reuse-Performance correlation. TaskInsight combines data reuse classification with performance and L3 misses to explain scheduling effects at the shared cache.

Similarly to the single-threaded case, there is a noticeable difference in how the two schedules touch new data: naive is touching the data used by all the tasks in the first 38 co-running sets (152 tasks), while smart is bringing new data into the caches throughout the entire execution. In addition, the smart schedule has 11% more shared-last reuses (dark blue area) than the naive schedule (50% vs 39%), meaning that the co-running tasks are using 11% more data from the previously executed tasks, thereby increasing the likelihood of hitting in the shared cache.

The figure also displays the results of the hardware performance counters in the bottom. The filled area (green) represents the absolute L3 miss ratio for each of the co-running set of tasks, while the line (green) shows the averaged performance (CPI) across all cores per co-running set of tasks. With both the TaskInsight data classification and the hardware performance counter results, we can now see the correlation between the classification of where a schedule’s tasks are reusing data from and the performance impact due to the resulting cache behavior. As all the data does not fit entirely in the shared cache, when the naive schedule touches all the data upfront, it results in a significant increase in the number of cache misses, and a corresponding decrease in performance for the co-executed tasks (x-axis 43 to 93).
It is also possible to see how an increase in \texttt{shared-last} reuses results in a lower L3 miss ratio, and vice-versa, where an increase in \texttt{older-reuses} generally results in more L3 misses. Two examples of this are highlighted in the figure, where closer reuses result in fewer cache misses, and farther reuses miss in the cache due to eviction. The \texttt{smart} schedule achieves a 20\% better L3 miss ratio than the \texttt{naive} schedule, yielding a 10\% performance improvement. It is only now with TaskInsight’s analysis that we can see what tasks are involved, when their data was reused, why a schedule impacted performance if it was beyond the cache size limits.

To demonstrate the value of this new analysis, we look at a significant performance change across schedulers and use our analysis to understand what scheduling decisions led to it. The significant change in the L3 miss ratio in Figure 39 (top, A) corresponds to the 86th set of co-running tasks. This set contains tasks 324, 430, 492 and 619, all of which are of type \texttt{dgemm}. From the figure we can see that combined they give a miss ratio of 66\%. Figure 40 shows the relative change in CPI and cache behavior for these tasks compared to the average of all instances of this task type in the schedule. For these specific co-executed tasks we see a 13\% increase in the private L2 misses and an 80\% average increase in L3 misses.

When examining where these task instances were executed in the \texttt{smart} schedule (Figure 39 bottom, B), we can now see that the source of the problem does not only come from the amount of reuse these tasks have from their previously executed tasks, but also from how they interact at the last level cache. From our analysis, we can compute that the size of the combined datasets of these four task instances is a total of 42K cachelines (roughly $2.7\text{MB}$). On the
other hand, when these tasks were executed in the smart schedule, they were overlapped (co-run) with other tasks whose combined datasets were not more than 32K cachelines (2MB). As a result, the smart schedule enabled these tasks to keep their working sets in the L3 cache, while the naive schedule did not. For instance, task 324 was executed in co-running set 22 in smart.

This analysis shows that our technique can be used to identify which scheduling decisions result in different memory behavior across tasks. Not only does it give insight into where the data is reused over time, but it can also determine the amount of shared and non-shared data at any point in time, which is vital information to understand if the combined datasets of the co-running tasks fit in the cache, and therefore the effects of scheduling at the shared cache level.

III.6 Implementation

Figure 44 shows an overview of how TaskInsight combines memory accesses and hardware performance counter information through a profiler, an instrumentation library and an analysis tool. The memory access profiling tool uses Pin [87] to sample the tasks’ memory accesses. While profiling, an address map is created, between each accessed address (at a cacheline granularity) and all tasks that use it. Note that this also captures execution order (schedule), and therefore provides all necessary information for the classification of the memory accesses.

Next, an instrumentation library collects per-task performance information. The library intercepts runtime calls (OmpSs [86] in our implementation) and records the hardware performance counters at the start and end of each task. The performance counter instrumentation must be run for both schedules, as the hardware results are schedule-dependent, and cannot run at the same time as the memory access profiler due to its performance overhead.

Finally, the analysis tool uses the memory accesses profile, the data from the hardware counters, and the schedule (per core in the multi-threaded case). It builds a data reuse graph, connecting datasets and tasks to describe the applications’ data characteristics, independent of the schedule. Each node in the data reuse graph 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. Combined, these allow us to understand how data is reused for any schedule, and it enables TaskInsight to reconstruct the datasets of any co-running task set.

With the per-core schedule as an input, it is possible to walk through the data reuse graph, analyzing each task’s dataset and comparing it to the previous tasks. While the graph is very dense, it is only necessary to walk it according to the input schedule. Furthermore, since the representation is schedule agnostic, it is possible to walk the same graph following a different schedule.
and generate an analysis for that schedule. This enables the prediction of the sharing under different schedules without the need to re-profile the application.

The TaskInsight methodology outlined here is general, transparent to the applications and independent of the runtime system used, making it directly applicable to any other task-based environment.

### III.7 Related Work

Previous work has proposed different ways to diagnose scheduling anomalies by either interactively visualizing information [85, 80, 88] or by simulating the task execution in order to provide a deterministic behavior of the scheduler [92, 83] without evaluating the performance behavior as a result of how memory is used. Significant work has been done to study the locality as a metric to characterize the workload of an application [94, 82] without considering the scheduling decisions taken as a result of the complex architectures, and only looking at the overall impact of locality on performance [89]. 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 through the execution and on how scheduling decisions of a task-based application affect the performance of task instances.

Drebes et al. [85], as well as other visualization tools ([80], [88]) 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. 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. [92] as well as Chronaki et al. [83] rely on simulation of the tasks’ execution in order to isolate the scheduler’s effect on performance from tasks’ unpredictable behavior. In our work we execute the entire application on real hardware and we characterize 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. [93] observed that tasks of the same type have different execution 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
TaskInsight consists of a profiling tool, an instrumentation library and an analysis tool.

Weinberg et al. [94], as well as Cheveresan et al. [82], propose using memory reuse as a metric to characterize workloads. Through this technique they analyze spatial and temporal locality of the application independent of the architecture. In our work we analyze the performance variation of an application when facing dynamic scheduling adaptations on modern architectures. By understanding how the memory is reused through the execution of the application we can evaluate if the scheduler is taking the correct decisions. Unlike [89] we evaluate the reused data throughout the execution and provide an analysis over time. This information can be used by an automatic tool to optimize the performance of the application.

III.8 Conclusion

In this work we presented TaskInsight, a methodology that provides high-level, quantifiable information that ties task scheduling decisions to how tasks reuse data and the resulting task performance. By combining schedule independent memory access profiling (to classify how data is reused between tasks) and schedule specific hardware performance counter data (to determine performance on a given system) we are able to identify which scheduling decisions...
impact performance, when they happen, and why they cause a problem. TaskInsight goes beyond previous work which typically used aggregate metrics to look at overall memory system behavior or ignored the task-level performance variation due to cache/scheduler interactions. With this deeper insight we can enable future generations of developers and schedulers to better optimize their applications for complex memory systems.

TaskInsight not only gives insight on how a scheduler can be improved but also an explanation for why tasks of the same type can demonstrate significant variation in performance (up to 60% in our examples). As a result, programmers can now quantitatively analyze the behavior of the scheduling algorithm and the runtime can use this information to dynamically make better decisions. TaskInsight diagnoses task scheduling misbehavior for both sequential and parallel applications, and we demonstrated how to use it to understand native multi-threaded executions which expose differences above 10% in performance due to 20% difference in reuses through the private caches and up to 80% difference in reuses through the shared last level cache, caused by scheduling.

By providing this insight into the coupling between the schedule’s behavior, data reuse through the cache hierarchy, and the resulting performance, we lay the groundwork for improving scheduling policies.

We are particularly interested in using this information to optimize for locality in NUMA aware architectures, bandwidth in CPU/GPU architectures, memory footprint or even energy efficiency.

III.9 Acknowledgments

We thank the reviewers for their feedback. This work was supported by the Swedish Research Council, the Swedish Foundation for Strategic Research project FFL12-0051 and carried out within the Linnaeus Centre of Excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center. This paper was also 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.
References


Paper IV
Paper IV

Analyzing Performance Variation of Task Schedulers with TaskInsight

Germán Ceballos, Thomas Grass, Andra Hugo and David Black-Schaffer

In Parallel Computing Journal (PARCO)
(to appear 2018)
Abstract
Recent scheduling heuristics for task-based applications have managed to improve performance by taking into account memory-related properties such as data locality and cache sharing. However, there is still a general lack of tools that can provide insights into why, and where, different schedulers result in different memory behavior, and how this is related to the applications’ performance.

To address this we present TaskInsight, a technique to characterize the memory behavior of different task schedulers through the analysis of data reuse across tasks. TaskInsight provides high-level, quantitative information that can be correlated with tasks’ performance variation over time to understand data reuse through the caches due to scheduling choices. TaskInsight is useful to diagnose and identify which scheduling decisions affected performance, when were they taken, and why the performance changed, both in single and multi-threaded executions.

We demonstrate how TaskInsight can diagnose cases where poor scheduling caused over 40% difference on average (and up to 7x slowdowns) across the Montblanc benchmarks due to changes in the tasks’ data reuse through the private and shared caches. This flexible insight is key for optimization in many contexts, including data locality, throughput, memory footprint or even energy efficiency.

IV.1 Introduction
Scheduling tasks in task-based applications has become significantly more difficult due to overall system complexity, and, in particular, deep, shared memory hierarchies. Typical approaches for optimizing scheduling algorithms consist of either providing an interactive visualization of the execution trace [85, 80] or simulating the tasks execution to evaluate the overall scheduling policy in a controlled environment [92, 83]. The developer then has to analyze the resulting profiling information and deduce if the scheduler behaves as expected, and qualitatively compare different schedulers.

Poor scheduling decisions often cause performance variations across tasks of the same type, which makes it hard to identify the root cause from the overall schedule. Scheduling strategies are sometimes implemented to be aware of these variations and ensure good load-balancing. However, understanding the underlying causes of individual tasks’ performance anomalies as well as the snowball effect of the dynamic scheduler is still an open question.

The effects of poor scheduling decisions can be most easily seen in idle execution time due to load imbalance from the inability to prioritize tasks on the critical path or appropriately map tasks to processors. However, scheduler decisions also impact data locality in the cache hierarchy by changing the order of producer and consumer tasks. The result of these decisions is performance variation across task, which can only be understood by analyzing how the tasks share data and how the schedule affects that sharing.

Generally, task-based application developers blame this performance degradation on data locality and attempt to characterize their workload based on data reuse without considering the dynamic interaction between the scheduler and the caches [94, 82]. This is simply because there has been no way to date
to obtain precise information on how the data was reused through the execution of an application, such as how long it remained in the caches, and how the scheduling decisions influenced this reuse. Without an automatic tool capable of providing insight as to whether and where the scheduler misbehaved, the programmer must rely on intuition to understand and adjust the scheduler for improved performance.

In this paper, we present TaskInsight, 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 through the cache hierarchy over time. We show how the reuse of data throughout the execution can provide insights into the performance of the scheduler, regardless of whether it is optimized for data locality, bandwidth, memory footprint, etc. Further, TaskInsight can interface directly with the task-based runtime system to provide this information both to the programmer and the scheduler.

Scheduler optimization is a notoriously difficult problem as past decisions affect choices and performance in the future, making it hard to explain performance without a detailed view across the program. Previous work [89] has shown the effects of data reuse distances in performance degradation. Those results were based on aggregated statistics and do not provide the necessary detail to manually (developer) or automatically (runtime system) adjust the schedule to improve performance or locality.

In order to understand the performance of a particular schedule, and thereby the scheduler itself, it is necessary to address three critical questions:

- *(Q1) What* scheduling decisions influenced the performance of the execution?
- *(Q2) When* did those decisions happen?
- *(Q3) Why* did those decisions affect the performance?

Answering these questions is vital for dynamic scheduling strategies that adjust their decisions in real time based on how tasks use the hardware resources. Scheduling decisions need to take into account the individual task performance to optimize the overall application, which is nearly impossible without answers to the above questions. The TaskInsight methodology shows how data reuses between tasks can provide key information for answering these questions, by enabling us to quantify their effects over time, and thereby exposing the interactions between the tasks’ performance and their schedule.

We make the following contributions:

1. A novel classification of the data of each task based on when the data is used over time. This classification is able to expose different memory behaviors inherent to the schedule.
2. A new analysis of schedulers based on their effects of the applications’ *temporal locality*, by connecting our classification to the measured performance results and statistics from the private caches.
3. A new technique to analyze schedulers based on the preservation of spatial locality of the data through time, by linking our classification with performance results and statistics from the shared caches.

We first investigate the impact of scheduling on memory behavior and performance by looking at its effects on a range of benchmark application configurations and schedulers (Section IV.2). From this study, we distinguish categories for benchmarks that are sensitive and non-sensitive to scheduling by looking at the performance differences (in task cycles, e.g., not including scheduler overhead) and L2/L3 cache misses among the benchmarks.

We then select a representative configuration of the benchmark Cholesky Factorization to present our tool, demonstrating how the overall performance of an application changes when executing with different schedules due to an increase in last-level cache misses (Section IV.3). This example motivates TaskInsight’s data classification technique, as it enables us to clearly differentiate the schedules in terms of their data reuse patterns, using a data reuse graph as in [81] (Section IV.4).

With this motivation, we extend the analysis and show how to connect the TaskInsight classification to changes in data reuse, changes in cache misses and changes in performance during the execution: first from the perspective of the private caches (temporal locality on a single-threaded execution, Section IV.5) and later including the shared caches (spatial locality on multi-threaded run, Section IV.6). This complete analysis is then used to demonstrate how TaskInsight enables us to understand other behaviors across the benchmarks and schedulers.

IV.2 Motivation

It is well known that cache optimization is crucial for performance, but real-world applications expose different sensitivities to changes in memory behav-
ior. Furthermore, task-based applications can vary wildly in their behavior based on several factors such as the size of the input problem (total data), the number of tasks they spawn (chunk size), how they distribute work among those tasks (parallelism), and how many tasks they can run in parallel (dependencies). Each of these factors can typically be controlled, either by configuring the application or the runtime. Nevertheless, it is often difficult to know which one is the best and how would the scheduler behave with these new configurations.

Before diving into understanding how scheduling changes the internal application’s memory behavior, it is worth studying how significant the effects of scheduling can be. To do that, we begin by looking at OmpSs[86] implementations of the Montblanc Benchmarks [84]. These benchmarks were designed to cover a diverse range of task-based applications and behaviors. We selected 9 benchmarks from the suite, and executed them with over 100 different combinations of input parameters (different input datasets, number of tasks, steps, dimensions, simulations, particles, etc.). We will call the combinations as configuration. Each configuration was executed with two different scheduling policies, one which attempts to optimize for locality (smart) and one which does not (naive). We will refer to the two runs as one experiment.

To understand how sensitive the benchmarks are to scheduling, we show how much of an impact the two schedulers have on the tasks’ performance in Figure 29. This gives us a distribution of the performance differences on a per-benchmark basis. As a result, we can see which benchmarks experience the largest sensitivity to scheduling across different configurations, and how often these performance differences occur.

The graph shows the percentage of the population (of the 100 configurations per benchmark, y-axis) as a function of slowdown (percentage between the two schedulers, x-axis). In essence, this summarizes how many of the experiments had a slowdown larger than X% (i.e. when changing the scheduling policy). From this we can see that benchmarks such as fft, cholesky, reduction and n-body experience a high variation in performance across a significant number of their configurations: 90% of the executions of fft had more than 60% performance difference when changing the scheduling policy; for reduction 30% of the executions had over 30% performance difference; and for cholesky, 40% had differences of over 30%.

On the other hand, benchmarks such as raytrace, dense-matrix and atomic-montecarlo-dynamics showed negligible performance changes between the schedulers across all configurations. Interestingly, being insensitive to scheduler changes appears in two forms depending on whether the performance impact is small or the number of configurations that experience it is small. raytrace (green line) shows that there is variation when chang-

\[\text{Note that this metric looks at the task execution time to reveal memory system effects, and therefore excludes scheduler overhead.}\]
Figure 30. Average (L2 cache misses, L3 cache misses, performance) change across all experiments of the same benchmark when changing the scheduling policy. E.g. 3d-stencil exposed 40% performance difference on average across all experiments, 170% more L3 cache misses and 116% more L2 misses. atomic-montecarlo had less than 1% performance difference on average and mainly due to a 19% increase in L2 cache misses. cholesky had 9.7% performance difference on average and mainly due to a 37% difference in L3 cache misses.

The results of comparing schedulers in Figure 29 shows us that there is significant variation in how applications respond to changes in scheduling, and a significant potential to improve performance with better scheduling. However, today’s tools and techniques do not enable us to analyze and understand how they are related to scheduling, why they occur, and how to fix or avoid them.

TaskInsight characterizes performance differences caused by changes in the memory behavior of the application due to scheduling. Figure 30 shows for each benchmark, the mean difference in total task cycles (performance), L2 cache misses and L3 cache misses across all benchmark configurations when changing the scheduling policy in almost every execution, however, all those differences are less than 1%, and a similar (but more noticeable) effect is observed in dense-matrix, with differences under 13%. On the other hand, atomic-montecarlo-dynamics shows that variation is very unlikely to occur (1 in 100 configurations), but when it does, the performance difference is significant (over 90%). These are two opposite and very extreme behaviors, but in the context of this work they have the same implication that there is little to be gained from optimizing the scheduler. For simplicity, we will classify both types as benchmarks that are performance insensitive to scheduling changes.
changing the scheduling policy. This adds further information to the previously observed categories: cholesky and 3d-stencil showed significant variation in the number of L3 cache misses, while nbody and fft see an increase in the number of L2 cache misses.

Other benchmarks such as atomic-montecarlo, dense-matrix, and raytrace did not show large variation in performance (cycle count), even though they have a fair increase both in L2 and L3 cache misses. These results allow us not only to determine benchmarks sensitive to scheduling, but also to see if sensitivity is likely to be caused by changes in memory behavior, and if so, which parts of the memory hierarchy are likely to be involved.\(^8\)

To build the foundation for TaskInsight, we first select and study a particular execution of one the Cholesky Factorization benchmark, as our data shows that its performance variation across schedulers is likely due to an increase in last-level cache (L3) misses.

### IV.3 Motivating Example

As we have seen in the previous section, different applications expose a variety of sensitivities to scheduling. The primary goal of TaskInsight is to be able to detect why there was a performance difference and when the scheduling decisions that led to that change happened. To understand how we can gain insight into these questions, we begin by looking at single-threaded runs of the cholesky factorization benchmark with a 32MB input matrix (256x256 block size), which enables us to hold one task’s dataset at a time in the 2MB last level cache. The application generates a total of 120 tasks of four different types (gemm, potrf, syrk, and trsm). For these initial experiments we use the TaskSim [91, 90] simulator, which allows us to precisely control the effects of memory bottlenecks. Results from native runs and multi-threaded executions are presented later.

Figure 31 shows the total cycle count (execution time of the tasks), total number of last-level cache misses and average task last-level cache misses-per-kilo-instruction (mpki), for the cholesky benchmark, with two different schedulers on OmpSs. The first scheduler, naive, uses a breadth-first-search policy, scheduling tasks in creation order. The second, smart, schedules tasks according to a heuristic, wherein child tasks are prioritized over the next task in a breadth-first order. The smart heuristic attempts to optimize for locality, as child tasks are likely to reuse data from their parents.

Note that we are reporting performance in terms of task cycle count. This metric counts just the time it takes to execute the tasks, and ignores overheads for the scheduler, runtime system, and load imbalance. By measuring just the task cycle count, we can accurately capture changes in the performance of

\(^8\)Note that TaskInsight does not model the effects of prefetching, so applications may see no decrease in performance despite an increase in cache misses if they are prefetcher-friendly.
specific tasks, which is essential to understand the impact of the scheduler on individual task performance.

As we see from the total cycle count, smart is 6% faster than the naive scheduler for cholesky. From the cycle count breakdown we see that the main difference comes from the number of cycles spent on DRAM accesses as a result of a 5% increase in last-level cache misses. This results in a 14% increase in the average task MPKI. However, while these statistics clearly show that the memory behavior and performance are affected by the scheduling choice, the overall statistics are not detailed enough to provide actionable insight into why we have this increase in cache misses, when they happen in the schedule, or what potential there is for improvement.

This analysis is typical of existing tools, which analyze the performance of the overall application [83] or of the different tasks independently [85]. However, they lack information on how the tasks influence one another through the execution (schedule and caches) and how the algorithmic decisions of the scheduler can impact the performance of the tasks, and thus of the application.
IV.4 Through the data-reuse glass

Tasks operate mostly on their own private data sets, but, over time, parts of tasks’ data may be reused by later tasks. This is typical of producer-consumer tasks and tasks which operate on a shared input. As a result, a portion of the data set is shared between multiple tasks. If the scheduler can arrange to execute the tasks close enough together in time, it will increase the chance that the shared data remains in the cache, and thereby improve performance through temporal locality.
Figure 34. Relative data reuse. Data reuse for smart schedule: new data is brought more sparsely in smaller chunks, and data is reused sooner showing more last-reuses.

To understand the impact of scheduling decisions on task data sharing, it is first necessary to analyze how much shared and private data each task has. TaskInsight does this by profiling the execution to sample the memory addresses for each task. Once the profile is collected, TaskInsight makes the following classification: For a given schedule, every memory access for a task is either new (the first time any task has touched the data) or reused from a previous task.

Figure 32 compares the cumulative amount of data touched as the program executes across the two schedules from Figure 31. 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 Figure 32 we can see how the naïve schedule (red curve) executes tasks in a way that touches new data much more aggressively, in bursts. On the other hand, the smart schedule (orange curve) is smoother, meaning that new data is brought at a lower rate. The flat regions on the curves indicate reuse-periods, wherein the scheduled tasks operate on previously used data, and therefore do not bring in any new-data. From this analysis it is clear that the different schedules result in tasks reusing data in significantly different patterns, which will therefore cause different cache miss rates, and impact performance.

Although the new-data category intuitively exposes the rate at which the applications install new data in the caches, it does not explain how the shared data is used. Thus, to understand the details of how the two schedules reuse data, we need to divide the memory behavior into more detailed categories:

- **new-data**: the first time the memory address is used in the application.
• last-reuse: the memory address was used by the immediately previous task before the current one.
• 2nd-last-reuse: the memory address was used by the second-to-last task before the current one.
• older-reuse: the address was used by a task that came more than two tasks previous to the current one.

Figures 33 and 34 show the breakdown of memory accesses (new-data, last-reuse, 2nd-last-reuse, older-reuse) for both schedules (naive, smart) 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 lower-middle region shows the percentage of second-last memory accesses, and finally the upper-middle 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 policy, compared to the naive approach, which touches most new-data during the first 16 tasks. In addition, the area corresponding to last-reuses increases considerably (more dark blue) in the smart schedule, meaning that more shared data is being reused sooner. This is also observed between tasks 100 to 115: in the naive schedule, most of the data used is coming from the second-last predecessor, but in the smart schedule, data is coming from its immediate predecessor. As the immediate predecessor is more recent, one would intuitively expect that this schedule would result in a higher cache hit rate and better performance.

With this classification, it is now clear that the two schedules have very different reuse characteristics. However, we need to translate the observations from the previous figures into relevant metrics to compare the schedules overall. TaskInsight uses aggregated statistics from each reuse category over time to understand how reuses flow from one category to another, as shown in Figure 35. Here we see the percentage of memory accesses corresponding to each category (y-axis, %), as a function of time (x-axis, by task number), for both the naive (left) and smart (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 scheduler has 11% more last-reuses than naive. 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 schedule turning 2nd-last-reuses into last-reuses, while the remaining 6% comes from improving older reuses.

Figure 35 not only allows us to see what aspects of the schedules are different, but also to precisely detect when the schedules have differences in data reuse. Since the tasks are uniquely identified, it is possible to point to the specific tasks that benefited from the rescheduling or were hurt by it. In the following section, we show how to connect this classification with performance measurements to also understand why performance was affected by changes in memory behavior due to scheduling.
Overall, the TaskInsight analysis allows us to understand the impact of scheduling changes in a way that can be used to improve performance by increasing reuse through caches. Approaches that only measure the actual cache miss ratios per task (e.g., 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 novel methodology enables scheduler designers to gain insight into how specific scheduling decisions impact later tasks.

IV.5 Analyzing Performance

The classification described in the previous section allows us to characterize the impact of different schedules on memory behavior in a hardware-agnostic manner. However, comparing the relative differences between these metrics is not enough to predict how they will affect performance. To accomplish this TaskInsight combines the data reuse classification (new, last, etc.) with performance measurements to explain changes in performance due to changes in data access.

We will first use TaskInsight to study how scheduling affects performance due to changes in the data’s temporal locality, and how it is related to cache reuse. To illustrate the analysis, we consider the simulated single-threaded execution of the Cholesky factorization from Section IV.2, which exposed a performance difference between two schedules. Focusing on single threaded execution allows us to exclude contention for the shared last-level cache. Section IV.6 later extends TaskInsight to analyze multi-threaded runs, both regard-
ing temporal and spatial locality changes due to scheduling, and explaining performance changes.

During the Cholesky factorization execution, we measure the performance of each task (average cycles-per-instruction, CPI) and cache behavior (last-level cache misses and accesses). The CPI for each task instance (color-coded by type) is shown in the bottom of Figure 35. We can see that the performance across tasks (CPIs) varies far more in the naive schedule (average 20% worse). For instance, if we look at Task ID 57, in the naive schedule it was executed as the 57th task, executing at 0.33 CPI. In the smart schedule, this same task was executed as the 32nd task, and ran 15% faster, at 0.29 CPI.

With TaskInsight we can analyze where the data is coming from for task 57 and correlate its CPI with its data reuse classification. This shows us that with the naive scheduler the task is reusing 98.1% of its data from older tasks (highlighted in the figure). Scheduling this task sooner with the smart scheduler results in 96.5% of the data is coming from the previously executed task, increasing the likelihood that the data is reused through the cache, and thereby increasing performance.

Moreover, we also see variations in task performance within the same schedule for tasks of the same type: e.g. with the smart scheduler tasks 113-118 are all syrk with same input size, however, task 113 has a 7% worse CPI than its subsequent ones. Again, by correlating with the new-data graph, we see that this is because it is bringing 20% new data for that schedule.

In addition, 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 essential: if data is never going to be touched again, it is possible to schedule a task that brings new data into the cache; if a task’s data is not going to fit in the cache, it 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 (size), as the amount of reuse that can be realized through the cache depends on the size of the tasks’ data.

IV.6 Multi-threaded Executions

The previous section showed how TaskInsight can quantitatively characterize different schedules with regards to their temporal data locality and data reuse through the caches for single-threaded applications.

When running multi-threaded applications, the complexity of the analysis is significantly increased by having multiple schedules executing in parallel across the cores and the effects of shared caches, which can cause the schedules to interfere with each other. The shared cache effects can be particularly
Figure 36. Performance-L2 correlation. Tasks from the same type are clustered better in the smart schedule, meaning less performance variation within the same type due to memory.

Figure 37. Performance-L3 correlation. Tasks and types exposing drastic performance degradation due to sensitivity to the shared cache. The smart schedule shows tasks with 50% less misses and 50% less performance variation within the same type.

important for performance as the cost of an L3 miss is much higher than an L2 miss (and L3 hit).

When using TaskInsight for multi-threaded executions, it provides information about the changes in temporal locality on a per core basis. However, to handle parallel executions of multiple schedules it is also necessary to understand the effects in performance of the shared cache.

As multi-threaded executions can execute tasks at a far higher rate, it is often difficult to identify tasks whose performance was affected by the schedule. We begin by showing how to use performance-to-memory correlation as
a filtering technique (Section IV.6.1). We then explain how to use TaskInsight on a per-core basis to reveal where schedules fail to preserve temporal data locality through the private caches (Section IV.6.2). Finally, we show how to extend TaskInsight to explain the effects of shared caches in performance variation, by also modeling how the schedules would share the data over time when co-running on different cores (spatial locality) (Section IV.6.3).

IV.6.1 Performance-memory filtering

For our analysis of parallel tasks we now consider a larger execution of the same Cholesky factorization on a quad-core machine\(^9\). The execution consists of 796 task instances in total, working on a 128MB matrix with block sizes of 256x256 elements, where 550 are of type \texttt{dgemm} (80% of the total cycle count), 114 are type \texttt{dsyrk}, 117 are type \texttt{dtrsm} and 15 are type \texttt{dpotrf}. Each of these task routines are from the Intel MKL library.

We executed the application with the same two scheduling policies as before (naive and smart) on 4 cores\(^10\), and collected data from hardware performance counters including number of instructions, cycles, and L2/L3 cache misses/accesses, per-task (implementation details in Section IV.8).

When aggregating these results, we see that the total cycles speedup of \texttt{smart} vs. \texttt{naive} is over 10%, and when looking at the cycle breakdown, \texttt{naive} incurs 9% more L2 misses on average per core, and 40% more L3 misses than \texttt{smart}. This demonstrates a significant variation due to scheduling.

TaskInsight can determine which specific tasks were affected by the schedule differences and at which specific moments in time. However, displaying data from a large multi-threaded execution would require very complex graphs for each execution core. To see the effect of scheduling and task type on memory related performance, we start by correlating performance to memory on a per task basis in Figures 36 and 37. This presentation allows us to identify which task instances had the most performance variation between the different schedules. These figures show scatter plots of per-task CPIs vs. (private) L2 miss ratios and (shared) L3 miss ratios, respectively, colored by task type for both schedules.

The performance-to-L2 miss ratio data (Figure 36) shows which tasks are sensitive to temporal locality changes through the L2 (sensitivities to the shared L3 are discussed in Section IV.6.3). As we can see, the variance in the performance of the tasks using the \texttt{naive} schedule (Figure 36 left) is significantly higher, especially for task types such as \texttt{dgemm} (blue) and \texttt{dsyrk} (red). When we compare this to the \texttt{smart} schedule (Figure 36 right), we see that

\(^9\)Intel Core i5-3550 CPU (Ivy Bridge), 8-way associative L1 cache, 256kB 8-way associative L2 cache, 6MB 12-way associative L3 cache, 16GB RAM.

\(^10\)One thread was pinned per core, and HyperThreading was disabled.
dgemm (blue), which represents 80% of the total execution time, have over 11% better CPI as a result of an 8% lower L2 miss ratio. This is seen in how the dgemm cluster moves down (lower CPI, better performance) and to the left (lower L2 miss ratio) between the naive (left) and smart (right) plots in Figure 36. A similar effect of the smart scheduler can be seen for the dsyrk (red) task cluster.

The opposite effect can be seen, though, for the dtrsm tasks (light-green). In the smart case, the cluster shows 4% more L2 misses, causing performance to be 6% worse. Despite this, the overall performance is still better for this schedule as the dtrsm tasks represent only 8% of the total execution time. In this case the scheduler has effectively traded off poorer temporal locality in the less frequent dtrsm for better locality in the far more frequent dgemm tasks.

From this correlation we can see both which task types are the most sensitive to memory effects due to scheduling, and also which tasks instances’ performance is most likely changed due to L2 caching effects and independent of caching effects. By using this filtering technique to identify these tasks, we can further study them with TaskInsight’s technique from Section IV.5 to fully understand if the changes in performance are due to reuses from L2, as we show in the following section.

IV.6.2 Temporal Locality of Private Caches

As each task is uniquely identified, it is possible to compare the difference of CPI and L2 miss ratio values between the two schedules on a per-task basis. This correlation allows us to distinguish between task instances without performance variation, task instances with performance variation due to non-memory effects (different CPI but no changes in L2/L3 miss ratios) and task instances with performance variation due to memory effects (both different in CPI and miss ratio). By filtering the analyzed tasks to the cases where performance variation is correlated with memory effects, we can target TaskInsight to study only on those tasks where we can benefit from scheduling insight as to how the schedule affected data reuse through the caches.

To analyze private caches, we can use the analysis from Section IV.5 on a per-core basis, as the private cache behavior is only affected by the local core’s schedule 11.

This enables the characterization of the two schedules in terms of how well they preserve temporal locality through the private caches. By only looking at the filtered tasks from the previous section, TaskInsight is able to show why those particular task instances missed more in the private L2 cache and when, which is essential to improve scheduling decisions.

11We are ignoring evictions due to the inclusive shared L3 cache as its size is far larger than the private L2s.
IV.6.3 Locality of Shared Caches

To understand how the shared cache affects the performance of different schedules, we extend TaskInsight to provide information regarding how tasks share the last level cache, and how this affects performance across different schedulers. The novelty of this approach lies on correlating caching, changes in schedule and changes in performance. This contribution is key to characterize not only how good a schedule is in temporal locality preservation through private caches, but also through shared caches, as well as characterize them in terms of spatial locality.

The correlations between task performance (CPI) and shared cache behavior (L3 Miss Ratio) is shown in Figure 37, colored by task type. For all task types, there is a rather linear correlation, meaning that the fewer L3 cache misses, the better the performance. Furthermore, we see that almost half of the dgemm tasks (blue) experience roughly twice the performance (lower CPI) for the smart schedule (right) over the naive schedule (left) with L3 miss ratio decreases of up to 50%.

Tasks that have a radical change in L3 misses are likely to have a performance loss, and vice-versa, so we consider those as candidates to study with TaskInsight.

While this performance-based analysis can identify which tasks saw worse performance due increases in L3 misses, it cannot give any insight into which scheduling decisions caused them. For example, in the naive schedule, two tasks with data reuse between them may have been scheduled further apart from each other, resulting in their data being evicted from the L3 before it could be reused. Alternatively, the tasks may have been executed very close to each other, but co-executed with other tasks that used a large disjoint dataset, thereby polluting the shared cache and evicting the data before it could be shared.

TaskInsight is able to explain where the difference in L3 cache misses is coming from, by modeling each set of co-running tasks, and then computing their predecessor’s reuse (temporal reuse) and amount of data reused at the same time (spatial reuse). This now allows us to determine which scheduling decisions cause the increase in miss ratios, which lead to worse performance.

To undertake this analysis, we first look at the sequence of tasks executed on each core, i.e. the per-core schedules. For each task instance, we determine the set of overlapping tasks for the schedule, which is the set of tasks executing on other cores at the same time. Figure 38 shows an example of this. In this case, the first co-running set comprises tasks 7, 45, 12 and 23, which are executed in parallel from core 0’s perspective. The second set contains tasks 8, 30, 13 and 31, and so on. This analysis gives us a sequence of co-running tasks over time, across all different cores. Each core will have a different sequence, but our analysis generates similar results for each of them.
Figure 38. Determining co-running sets: for each core, the set of co-running (overlapping) tasks is identified and used to build a single set with their memory addresses during analysis.

Figure 39. Reuse-Performance correlation. TaskInsight combines data reuse classification with performance and L3 misses to explain scheduling effects at the shared cache.

At the same time, TaskInsight builds the application’s (schedule-independent) reuse graph introduced in [81], and combines it with the co-running task se-
quence to compute the set of memory addresses used by each co-running set. This allows to model the sequence of co-running tasks over time, and use the analysis in Section IV.5 to analyze how much data was reused over time, but in the shared cache.

The result of this analysis is shown in Figure 39: the top half of the graph corresponds to the naïve schedule and the bottom half to the smart schedule. The x-axis represents execution order of the 216 sets of co-running tasks. The top of each graph shows the classification of the reuses provided by TaskInsight as in Figure 34.

Similarly to the single-threaded case, there is a noticeable difference in how the two schedules touch new data: naïve is touching the data used by all the tasks in the first 38 co-running sets (152 tasks), while smart is bringing new data into the caches throughout the entire execution. In addition, the smart schedule has 11% more shared-last reuses (dark blue area) than the naïve schedule (50% vs 39%), meaning that the co-running tasks are using 11% more data from the previously executed tasks, thereby increasing the likelihood of hitting in the shared cache.

The figure also displays the results of the hardware performance counters in the bottom. The filled area (green) represents the absolute L3 miss ratio for each of the co-running set of tasks, while the line (green) shows the averaged performance (CPI) across all cores per co-running set of tasks. With both the TaskInsight data classification and the hardware performance counter results, we can now see the correlation between the classification of where a schedule’s tasks are reusing data from and the performance impact due to the resulting cache behavior. As all the data does not fit entirely in the shared cache, when the naïve schedule touches all the data upfront, it results in a significant increase in the number of cache misses, and a corresponding decrease in performance for the co-executed tasks (x-axis 43 to 93).

It is also possible to see how an increase in shared-last reuses results in a lower L3 miss ratio, and vice-versa, where an increase in older-reuses generally results in more L3 misses. Two examples of this are highlighted in the figure, where closer reuses result in fewer cache misses, and farther reuses miss in the cache due to eviction. The smart schedule achieves a 20% better L3 miss ratio than the naïve schedule, yielding a 10% performance improvement. It is only now with TaskInsight’s analysis that we can see what tasks are involved, when their data was reused, why a schedule impacted performance if it was beyond the cache size limits.

To demonstrate the value of this new analysis, we look at a significant performance change across schedulers and use our analysis to understand what scheduling decisions led to it. The significant change in the L3 miss ratio in Figure 39 (top, A) corresponds to the 86th set of co-running tasks. This set contains tasks 324, 430, 492 and 619, all of which are of type dgemm. From the figure we can see that combined they give a miss ratio of 66%. Figure 40 shows the relative change in CPI and cache behavior for these tasks compared
to the average of all instances of this task type in the schedule. For these specific co-executed tasks we see a 13% increase in the private L2 misses and an 80% average increase in L3 misses.

When examining where these task instances were executed in the smart schedule (Figure 39 bottom, B), we can now see that the source of the problem does not only come from the amount of reuse these tasks have from their previously executed tasks, but also from how they interact at the last level cache. From our analysis, we can compute that the size of the combined datasets of these four task instances is a total of 42K cachelines (roughly 2.7MB). On the other hand, when these tasks were executed in the smart schedule, they were overlapped (co-run) with other tasks whose combined datasets were not more than 32K cachelines (2MB). As a result, the smart schedule enabled these tasks to keep their working sets in the L3 cache, while the naive schedule did not. For instance, task 324 was executed in co-running set 22 in smart.

This analysis shows that our technique can be used to identify which scheduling decisions result in different memory behavior across tasks. Not only does it give insight into where the data is reused over time, but it can also determine the amount of shared and non-shared data at any point in time, which is vital information to understand if the combined datasets of the co-running tasks fit in the cache, and therefore the effects of scheduling at the shared cache level.
IV.7 Detecting problems in other benchmarks

In the previous sections we analyzed how tasks’ data reuse changes led to performance variation in the cholesky application. In this section we show how to use the same TaskInsight analysis to characterize the memory behavior of the other benchmarks shown in Figure 29 and understand problems that caused performance variation. For this we have selected three of the scheduling-sensitive benchmarks: fft (solving a Fast Fourier Transform), reduction (non-trivial computation over vectors and reduction of the results) and histogram (computing the histogram of a sample population). The analysis will proceed as follows: We will first observe the performance variation over time to identify the main differences between the two schedulers (naïve and smart). We will then look at the profile data to see whether the source of the performance difference is memory, such as more L2 or L3 cache misses. Finally, we will correlate this data with TaskInsight’s reuse analysis to understand how this memory behavior comes from different data reuse patterns in the different schedules.

Figure 41. If data is used in a tidy pattern (smart, left) it fits in the cache, exposing a perfect reuse pattern and incurring much less cache misses. On the other hand, a more spread pattern (naïve, right) has constant cache misses across the entire execution with a performance penalty.
IV.7.1 reduction: a single task can have a severe impact if data is not in the cache.

Another interesting case study is reduction\textsuperscript{12}, where most of the performance difference comes from L2. Once again, data for L3 is not displayed because there is no relevant change across schedules.

From Figure 41 we can see that in smart, there is a cycle drop after executing the first 36 tasks, and it corresponds to a drop in L2 misses. From TaskInsight reuse analysis we see that data is still being reused, and very recently (many last-reuses), meaning that the data is present in L2.

On the other hand, naïve has a lower number of L2 misses, but spread across the entire execution, causing a more constant average cycle count. From the TaskInsight reuse analysis we see that data is also being reused recently, but in different fashion.

The interesting singularity we observe is task number 63, which brings a significant amount of new data. In smart, this task was scheduled in the very beginning (5th), and it is both touching a large set of new data and reusing very recent data.

In naïve, this task, because of dependency flexibility, was scheduled much later (63), and by that time, now the data it has to reuse is much older as we see from TaskInsight’s classification. Because its reuse is not much further away, the task generates roughly 10x more L2 misses (also 10x more L3 misses) and its execution now is 400X slower than in smart (note the Log scale in the top Cycle Over Time chart).

This is an interesting example of an application that exposes two fascinating effects. First, if data is used in a tidy pattern, it might fit in the cache, delivering good reuse and incurring many fewer cache misses, versus a more spread-out pattern that has constant misses across the entire execution. Second, a single task can suffer significant performance loss by being scheduled further away from tasks with which it shares data, thereby creating a significant penalty to the application’s overall performance. To understand these issues we needed TaskInsight’s data usage classification combined with the performance profile.

IV.7.2 histogram: differences in bringing new data sooner or later.

Another example where temporal locality of data at the private caches matters significantly is the benchmark histogram\textsuperscript{13}, Figure 42. The largest slowdown we observe is seen in a spike in the cycle count for naïve, due to an increase in L2 misses (creating a difference of over 20\% across the two schedules).

TaskInsight shows what the problem is (the reuse of old data and use of new data) and where it comes from (the 18th task). Task 18 was executed

\textsuperscript{12}inputs: nelem=8M, nt=128
\textsuperscript{13}inputs: nelem=4M, nbins=256, nt=128
as number 1 in \textit{smart}, and has a particularly large data set. By bringing its data in first, it enables that data to be installed in the cache and we see reuse by subsequent tasks. On the other hand, if this task is scheduled later, as in \textit{naive}, it will reuse many smaller portions of already executed tasks (with smaller data sets) which were very likely evicted from the cache. This causes a domino-effect that affects several tasks creating a substantial difference in the overall cycle count.

TaskInsight not only shows the importance of scheduling a task sooner or later because of its consequences, but it also allows us to detect which specific tasks had this type of behavior and what other tasks are affected, enabling new insight into the scheduler’s behavior.

IV.7.3 \texttt{fft}: Sudden reuses of old data result in spikes of last level cache misses (L3), hurting performance.

Despite most of the performance difference in \texttt{fft}\textsuperscript{14} coming from sensitivity to L2 cache misses, there is also an interesting effect worth studying in the L3. We start by applying TaskInsight’s analysis from Section IV.6 and displaying the co-running sets over time along with performance counter information.

As we see from Figure 43, both schedulers show two spikes with a sudden slowdown in cycles at co-running sets 35 and 73, which are correlated with spikes in L3 misses. The number of L2 misses is not displayed in this case, for simplicity, but the trend is similar to L3.

\textsuperscript{14}inputs: \texttt{nelem=}1M, \texttt{bs-tr=}64, \texttt{bs-fft=}128

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure42.png}
\caption{Scheduling bigger tasks sooner (\textit{smart}, left) will bring a larger amount of new-data sooner, exposing more last-reuses between subsequent tasks.}
\end{figure}
Figure 43. TaskInsight analysis of $\texttt{fft}$, where reuse of older data incurred in more L3 cache misses, causing two spikes in the tasks cycle count.

The two spikes in the total cycles (performance, top) align with an increase in L3 misses (middle). By looking at TaskInsight’s data classification (bottom), we can see that at those places where the application is missing more, it is also reusing older data, which is likely not present in the cache. By correlating this data over time, TaskInsight shows us that this cause is from the tasks that executed together at these times not be scheduled to reuse data effectively.

At the same time, by comparing the two schedules, we see that the second scheduler had over 10% worse overall performance. When looking at the cycle count over time and L3 misses, we see that during co-running sets 0 to 33, $\texttt{naive}$ incurs twice as many L3 cache misses, explaining where the overall performance difference is coming from.

Even though we can observe this performance and memory behavior directly with performance counters, is is only by adding TaskInsight’s data classification that we can understand that the extra cycles seen in $\texttt{naive}$ is only from new-data accesses which should be coming from main memory. This might seem counter-intuitive, as the reuse analysis is almost identical in both schedulers, but other memory factor (not yet included in this model) like prefetching fit as very good candidates responsible of the performance difference.

IV.8 Implementation

Figure 44 shows an overview of how TaskInsight combines memory accesses and hardware performance counter information through a profiler, an instru
mentation library and an analysis tool. The memory access profiling tool uses Pin [87] to sample the tasks’ memory accesses. While profiling, an *address map* is created, between each accessed address (at a cacheline granularity) and all tasks that use it. Note that this also captures execution order (schedule), and therefore provides all necessary information for the classification of the memory accesses.

Next, an instrumentation library collects per-task performance information. The library intercepts runtime calls (OmpSs [86] in our implementation) and records the hardware performance counters at the start and end of each task. The performance counter instrumentation must be run for both schedules, as the hardware results are schedule-dependent, and cannot run at the same time as the memory access profiler due to its performance overhead.

Finally, the analysis tool uses the memory accesses profile, the data from the hardware counters, and the schedule (per core in the multi-threaded case). It builds a *data reuse graph*, connecting datasets and tasks to describe the applications’ data characteristics, independent of the schedule. Each node in the data reuse graph 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. Combined, these allow us to understand how data is reused for any schedule, and it enables TaskInsight to reconstruct the datasets of any co-running task set.

With the per-core schedule as an input, it is possible to walk through the data reuse graph, analyzing each task’s dataset and comparing it to the previous tasks. While the graph is very dense, it is only necessary to walk it according to the input schedule. Furthermore, since the representation is schedule agnostic, it is possible to walk the same graph following a different schedule and generate an analysis for that schedule. This enables the prediction of the sharing under different schedules without the need to re-profile the application, as illustrated in Figure 45.

The TaskInsight methodology outlined here is general, transparent to the applications and independent of the runtime system used, making it directly applicable to any other task-based environment.

IV.9 Related Work

Previous work has proposed different ways to diagnose scheduling anomalies by either interactively visualizing information [85, 80, 88] or by simulating the task execution in order to provide a deterministic behavior of the scheduler [92, 83] without evaluating the performance behavior as a result of how memory is used. Significant work has been done to study the locality as a metric to characterize the workload of an application [94, 82] without considering the scheduling decisions taken as a result of the complex architectures, and only looking at the overall impact of locality on performance [89]. 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 through the execution and on how scheduling decisions of a task-based application affect the performance of task instances.

Drebes et al. [85], as well as other visualization tools ([80], [88]) 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. 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. [92] as well as Chronaki et al. [83] rely on simulation of the tasks’ execution in order to isolate the scheduler’s effect on performance from tasks’ unpredictable behavior. In our work we execute the entire application on real hardware and we characterize 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. [93] observed that tasks of the same type have different execution 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 differences, that is the way the tasks are reusing the memory, and we provide information that can be used by the scheduler in order to avoid such effects.

Weinberg et al. [94], as well as Cheveresan et al. [82], propose using memory reuse as a metric to characterize workloads. Through this technique they analyze spatial and temporal locality of the application independent of the architecture. In our work we analyze the performance variation of an application when facing dynamic scheduling adaptations on modern architectures. By understanding how the memory is reused through the execution of the application we can evaluate if the scheduler is taking the correct decisions. Unlike [89] we evaluate the reused data throughout the execution and provide an analysis over time. This information can be used by an automatic tool to optimize the performance of the application.
Figure 44. Methodology Overview: TaskInsight consists of a profiling tool, an instrumentation library and an analysis tool.

Figure 45. Profiling vs Instrumenting Multiple Schedules: The instrumentation phase has to be executed once per different schedule to collect information from hardware performance counters. However, data from memory accesses can be collected only once, as TaskInsight can model arbitrary schedules from a single profiling run, reducing overhead.

IV.10 Conclusion

In this work we presented TaskInsight, a methodology that provides high-level, quantifiable information that ties task scheduling decisions to how tasks reuse data and the resulting task performance. By combining schedule independent
memory access profiling (to classify how data is reused between tasks) and schedule specific hardware performance counter data (to determine performance on a given system) we are able to identify which scheduling decisions impact performance, when they happen, and why they cause a problem. TaskInsight goes beyond previous work which typically used aggregate metrics to look at overall memory system behavior or ignored the task-level performance variation due to cache/scheduler interactions. With this deeper insight we can enable future generations of developers and schedulers to better optimize their applications for complex memory systems.

TaskInsight not only gives insight into how a scheduler can be improved, but also an explanation for why tasks of the same type can demonstrate significant variation in performance (up to 60% in our examples). As a result, programmers can now quantitatively analyze the behavior of the scheduling algorithm and the runtime can use this information to dynamically make better decisions. We showed these capabilities for both sequential and parallel applications, and we demonstrated how to use them to understand native multi-threaded executions. Our analysis exposed scheduler-induced performance differences of above 10% due to 20% changes in data reuse through the private caches and up to 80% difference data reuse through the shared last level cache.

By providing this insight into the coupling between the schedule’s behavior, data reuse through the cache hierarchy, and the resulting performance, we lay the groundwork for improving scheduling policies. This type of analysis can be extended to include more complex effects and systems, such as prefetchers, NUMA/NUCA architectures, bandwidth in CPU/GPU architectures, memory footprint and even energy efficiency.

IV.11 Acknowledgments

This work was supported by the Swedish Research Council, the Swedish Foundation for Strategic Research [project FFL12-0051] and carried out within the Linnaeus Centre of Excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center. This paper was also published with the support of the HiPEAC network that received funding from the European Union’s Horizon 2020 research and innovation programme [grant agreement no. 687698].
References


<table>
<thead>
<tr>
<th>Year</th>
<th>Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017</td>
<td>001</td>
<td>Diana Yamalova: <em>Hybrid Observers for Systems with Intrinsic Pulse-Modulated Feedback</em></td>
</tr>
<tr>
<td>2016</td>
<td>012</td>
<td>Peter Backeman: <em>New Techniques for Handling Quantifiers in Boolean and First-Order Logic</em></td>
</tr>
<tr>
<td>2016</td>
<td>011</td>
<td>Andreas Svensson: <em>Learning Probabilistic Models of Dynamical Phenomena Using Particle Filters</em></td>
</tr>
<tr>
<td>2016</td>
<td>010</td>
<td>Aleksandar Zeljić: <em>Approximations and Abstractions for Reasoning about Machine Arithmetic</em></td>
</tr>
<tr>
<td>2016</td>
<td>009</td>
<td>Timofey Mukha: <em>Inflow Generation for Scale-Resolving Simulations of Turbulent Boundary Layers</em></td>
</tr>
<tr>
<td>2016</td>
<td>008</td>
<td>Simon Sticko: <em>Towards Higher Order Immersed Finite Elements for the Wave Equation</em></td>
</tr>
<tr>
<td>2016</td>
<td>007</td>
<td>Volkan Cambazoglu: <em>Protocol, Mobility and Adversary Models for the Verification of Security</em></td>
</tr>
<tr>
<td>2016</td>
<td>006</td>
<td>Anton Axelsson: <em>Context: The Abstract Term for the Concrete</em></td>
</tr>
<tr>
<td>2016</td>
<td>005</td>
<td>Ida Bodin: <em>Cognitive Work Analysis in Practice: Adaptation to Project Scope and Industrial Context</em></td>
</tr>
<tr>
<td>2016</td>
<td>004</td>
<td>Kasun Hewage: <em>Towards a Secure Synchronous Communication Architecture for Low-power Wireless Networks</em></td>
</tr>
<tr>
<td>2016</td>
<td>003</td>
<td>Sven-Erik Ekström: <em>A Vertex-Centered Discontinuous Galerkin Method for Flow Problems</em></td>
</tr>
<tr>
<td>2016</td>
<td>002</td>
<td>Rubén Cubo: <em>Mathematical Modeling for Optimization of Deep Brain Stimulation</em></td>
</tr>
</tbody>
</table>