Understanding Task Parallelism

Providing insight into scheduling, memory, and performance for CPUs and Graphics

GERMÁN CEBALLOS
Abstract

Maximizing the performance of computer systems while making them more energy efficient is vital for future developments in engineering, medicine, entertainment, etc. However, the increasing complexity of software, hardware, and their interactions makes this task difficult. Software developers have to deal with complex memory architectures such as multilevel caches on modern CPUs and keeping thousands of cores busy in GPUs, which makes the programming process harder.

Task-based programming provides high-level abstractions to simplify the development process. In this model, independent tasks (functions) are submitted to a runtime system, which orchestrates their execution across hardware resources. This approach has become popular and successful because the runtime can distribute the workload across hardware resources automatically, and has the potential to optimize the execution to minimize data movement (e.g., being aware of the cache hierarchy).

However, to build better runtime systems, we now need to understand bottlenecks in the performance of current and future multicore architectures. Unfortunately, since most current work was designed for sequential or thread-based workloads, there is an overall lack of tools and methods to gain insight about the execution of these applications, allowing both the runtime and the programmers to detect potential optimizations.

In this thesis, we address this lack of tools by providing fast, accurate and mathematically-sound models to understand the execution of task-based applications. In particular, we center these models around three key aspects of the execution: memory behavior (data locality), scheduling, and performance. Our contributions provide insight into the interplay between the schedule's behavior, data reuse through the cache hierarchy, and the resulting performance. These contributions lay the groundwork for improving runtime systems. We first apply these methods to analyze a diverse set of CPU applications, and then leverage them to one of the most common workloads in current systems: graphics rendering on GPUs.

Keywords: Task-based programming, Task Scheduling, Analytical Cache Model, Scheduling, Runtime Systems, Computer Graphics (rendering)

Germán Ceballos, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Germán Ceballos 2018

ISSN 1651-6214
ISBN 978-91-513-0485-4
urn:nbn:se:uu:diva-363924 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-363924)
Dedicated to the love of my life
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. All papers are reprinted verbatim, and have been reformatted to the one-column format of this book.
Other publications not accounted in this thesis:


Sampled simulation is a mature technique for reducing simulation time of single-threaded programs. Nevertheless, current sampling techniques do not take advantage of other execution models, like task-based execution, to provide both more accurate and faster simulation. Task-based programming models allow the programmer to specify program segments as tasks which are instantiated many times and scheduled dynamically to available threads. Due to variation in scheduling decisions, two consecutive executions on the same machine typically result in different instruction streams processed by each thread.

In this paper, we propose TaskPoint, a sampled simulation technique for dynamically scheduled task-based programs. We leverage task instances as sampling units and simulate only a fraction of all task instances in detail. Between detailed simulation intervals, we employ a novel fast-forwarding mechanism for dynamically scheduled programs. We evaluate different automatic techniques for clustering task instances and show that DBSCAN clustering combined with analytical performance modeling provides the best trade-off of simulation speed and accuracy.

TaskPoint is the first technique combining sampled simulation and analytical modeling and provides a new way to trade off simulation speed and accuracy. Compared to detailed simulation, TaskPoint accelerates architectural simulation with 8 simulated threads by an average factor of 220x at an average error of 0.5% and a maximum error of 7.9%.
Complex memory hierarchies of nowadays machines make it very difficult to estimate the execution time of tasks as depending on where the data is placed in memory, tasks of the same type may end up having different performances. Multiple scheduling heuristics have managed to improve performance by taking into account memory-related properties such as data locality and cache sharing. However, we may see tasks in certain applications or phases of applications that take little or no advantage of these optimizations. Without understanding when such optimizations are effective, we may trigger unnecessary overhead at the runtime level.

In previous work we introduced TaskInsight, a technique to characterize how the memory behavior of the application is affected by different task schedulers through the analysis of data reuse across tasks. We now use this tool to dynamically trace the scheduling decisions of multi-threaded applications through their execution and analyze how memory reuse can provide information on when and why locality-aware optimizations are effective and impact performance.

We demonstrate how we can detect particular scheduling decisions that produced a variation in performance, and the underlying reasons for applying TaskInsight to several of the Montblanc benchmarks. This flexible insight is a key for both the programmer and runtime to allow assigning the optimal scheduling policy to certain executions or phases.

In Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), held in conjunction with the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Austin, Texas, USA, February 2017.

Recent scheduling heuristics for task-based applications have managed to improve their 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 were they 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.
Contents

1 Introduction ................................................................................................ 15
  1.1 The Rise of Task-Based Programming ......................................... 15
  1.2 The Light and Dark Sides of Runtime Systems .................. 17

2 The Interplay Between Scheduling, Memory and Performance .......... 19
  2.1 Memory and Performance: A Well-studied Area .................. 19
  2.2 Scheduling as a Challenge ..................................................... 20

3 Understanding Tasks Parallelism in CPUs ............................................... 25
  3.1 Contribution 1: Task Pirate ....................................................... 25
  3.2 Contribution 2: StatTask ......................................................... 28
  3.3 Contribution 3: TaskInsight ....................................................... 31

4 Understanding Task Parallelism in Graphics Rendering ......................... 37
  4.1 Contribution 4: Behind the Scenes ......................................... 38
  4.2 Contribution 5: Tail-PASS ......................................................... 45

5 Conclusion .................................................................................................. 51

6 This Thesis in a Nutshell ........................................................................... 52

7 Svensk Sammanfattning ............................................................................ 57

8 Resumen en Español ................................................................................. 59

9 Acknowledgements .................................................................................... 61

References ........................................................................................................ 67
Abstract

Maximizing the performance of computer systems while making them more energy efficient is vital for future developments in engineering, medicine, entertainment, etc. However, the increasing complexity of software, hardware, and their interactions makes this task difficult. Software developers have to deal with complex memory architectures such as multilevel caches on modern CPUs, and keeping thousands of cores busy in GPUs, which makes the programming process harder.

Task-based programming provides high-level abstractions to simplify the development process. In this model, independent tasks (functions) are submitted to a runtime system, which orchestrates their execution across hardware resources. This approach has become popular and successful because the runtime can distribute the workload across hardware resources automatically, and has the potential to optimize the execution to minimize data movement (e.g., being aware of the cache hierarchy).

However, to build better runtime systems we now need to understand bottlenecks in the performance of current and future multicore architectures. Unfortunately, since most current work was designed for sequential or thread-based workloads, there is an overall lack of tools and methods to gain insight about the execution of these applications, allowing both the runtime and the programmers to detect potential optimizations.

In this thesis, we address this lack of tools by providing fast, accurate and mathematically-sound models to understand the execution of task-based applications. In particular, we center these models around three key aspects of the execution: memory behavior (data locality), scheduling, and performance. Our contributions provide insight into the interplay between the schedule’s behavior, data reuse through the cache hierarchy, and the resulting performance. These contributions lay the groundwork for improving runtime systems. We first apply these methods to analyze a diverse set of CPU applications, and then leverage them to one of the most common workloads in current systems: graphics rendering on GPUs.
1. Introduction

1.1 The Rise of Task-Based Programming

The arrival of parallel multicore architectures has forced mainstream developers to deal with the harder and more complex process of reasoning about parallel execution and resource sharing. Modern multicore architectures continue to become more complex: processors with multiple CPU cores of different sizes, hundreds of GPU cores sharing the same die, deep multi-level cache hierarchies, different memory network topologies, etc. As hardware parallelism increases, resources such as caches, busses and main memory are shared in ways that differ from CPU to CPU. This resource sharing can have significant impact on the performance of the applications, so doing it efficiently is a very important issue.

Because of this, optimizing application performance on these architectures now requires a deep understanding of application’s data usage, data sharing, and how it interacts with the hardware memory components. Legacy libraries and programming models such as Pthreads and MPI are adapting to these changes by providing new APIs and programmer’s support, but the scaling of these systems is still limited by the lack of understanding of performance bottlenecks. This has generated many new proposals on introducing new parallel programming models that increase the level of abstraction to simplify the reasoning, coding and debugging process.

One successful example of these new, high-level programming models is task-based programming. In task-based programming, an additional level of abstraction between the application and the system is inserted: a runtime system. The developer structures the application in a way that spawns small independent tasks (functions or units of code) and submits them to a runtime system, which queues them for execution. The runtime picks the next available task(s) according to a scheduling policy and assigns them to the appropriate resources (threads or physical cores).

There are several reasons why this paradigm has become successful:

1. **Tasks allow for a higher-level problem description**: With tasks, the programmer only needs to expose the dependencies between tasks, but not to reason about how to best resolve them. This eliminates the need for low-level execution organization and allows the programmer to express the problem at a higher level. In contrast, when using threads, the application needs to be structured in terms of physical threads to yield good efficiency. In addition, the best mapping to the physical threads is system specific, which makes it hard to optimize for performance.
Figure 1.1. High-level overview of a task-based system. The application interacts with a runtime system to submit tasks (A to E) using an API. The runtime system, aware of both the hardware architecture and the application information, will determine the schedule (execution order) of the tasks.

2. **Tasks can be automatically scheduled for performance:** In a task-based runtime, the task scheduler has access to higher-level information about dependencies, task types, task sizes, and the underlying system that can be used to guide the scheduling, trading off fairness for performance. The runtime also detaches program logic from architectural details, such as the cache hierarchy and the cache sizes. On the other hand, thread schedulers are usually agnostic to the thread workload or how the thread is performing. They tend to distribute time slices in a fair distribution (e.g. round-robin fashion), because it is the safest strategy when the scheduler is oblivious to the workloads’ performance sensitivities and requirements.

3. **Tasks simplify load balancing:** The runtime scheduler takes care of load-balancing the tasks, using the right number of threads and distributing work evenly across physical cores. In traditional libraries, such as pthreads, the programmer has to spawn the correct number of threads, and pin or migrate the threads manually.

4. **Tasks are lightweight:** Tasks are much faster to start and terminate. Current frameworks and implementations reported speeds up to 20 times faster than threads. This is a major advantage when running massively parallel applications with hundreds to thousands of tasks, since significant overhead is avoided.

These advantages have led to the development of many production-quality frameworks, including OpenMP tasks [41], OmpSs [21], StarPU [6], and In-
tel’s TBB. Figure 1.1 shows an overview of how these systems are typically organized. The application interacts with the runtime system though an API, detaching program logic from architectural details. This allows the runtime system to optimize the execution for better performance using both program and architecture information in a transparent way for the programmer.

1.2 The Light and Dark Sides of Runtime Systems

Delegating the scheduling and coordination of the tasks to the runtime system simplifies the coding process while also providing flexibility to adapt to multiple architectures in a transparent way. Using a runtime also allows to optimize for different goals. For example, minimizing energy might require a very different schedule than maximizing performance does. Most importantly, the runtime’s scheduler opens up the possibility of introducing new techniques to improve performance without having to change the application itself.

Although the runtime layer provides the possibility of automatically optimizing applications, it also incurs burdens, both in the development of applications and in their execution. First of all, the interfaces (APIs) provided by the runtimes should be expressive enough to cover a wide range of parallel applications.

Second, in a task-based application 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.

Lastly, introducing an extra layer between the application and the Operating System increases the execution overhead. For this approach to work, the performance benefits that the runtime can deliver need to exceed the overhead of the runtime management.

In light of the newer memory hierarchies and more complex systems, better runtimes are needed, capable of understanding this complexity to optimize performance. To build better runtimes, it is key to understand the fundamental interactions between tasks at the memory system level. In particular, (1) how resource sharing affects 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 chosen schedule and the application’s memory behavior impacts the overall performance or execution. We summarize these three interactions in the Scheduling-Memory-Performance triad, first introduced in [15] and depicted in Figure 1.2.

In this thesis, we investigate this triad across two application domains: CPU-based task applications, and GPU-based graphics rendering. We organize the thesis in two parts.

In the first part, we present tools and techniques for gaining insight into how these three key factors interrelate to each other, enabling both the runtime and the programmer to understand large-scale applications and potential
optimizations. Specifically, we study how scheduling affects the data locality properties of the applications, and therefore, how the performance is affected by the data placement and reuse throughout the caches. With these contributions it is possible to explain performance variation across different schedules for the same application.

In the second part, we apply these techniques to one of the world’s most popular task-based workloads: graphics rendering in tiled GPU architectures. We characterize graphics applications such as 2D and 3D games and animations, revealing potential performance optimizations for future memory systems.

These contributions provide powerful new insights into the complex memory interactions present in task-based systems and lay the groundwork for improving the task-based programming model, the runtime systems and the hardware.
2. The Interplay Between Scheduling, Memory and Performance

The Schedule-Memory-Performance triad (Figure 1.2) was first introduced in [15], with the intention to summarize the three key components of task-based programs executions.

The memory-performance edge of the triad represents the interactions between memory and performance. This area has been covered extensively by previous contributions, and we will discuss it in more detail in Section 2.1 as previous work. It has been shown that, most often, improving application data locality results in a positive impact on performance [17, 30, 8, 52, 32, 39]. In this context, several techniques have been proposed to understand how performance changes depending on how data is shared in parallel applications [8, 9, 23].

Traditional parallel programming models lacked intelligent runtimes, so memory-performance interactions were largely self-contained. Task-based programming adds scheduling, opening up new complexities and opportunities. Understanding tasks suddenly becomes harder, given that both other components (memory and performance) are affected by scheduling decisions.

Instead of studying the complex interactions of the triad altogether, we propose to study each edge individually to understand them better. For that, we will start by exploring previous state-of-the-art contributions: how they provide insight into this interplay, why they are limited in a task-based context and our plan to go beyond their limitations.

2.1 Memory and Performance: A Well-studied Area

When considering sequential or thread-based applications, there is a clear correlation between the achieved performance and their data locality. Most often, applications with memory access patterns that expose more spatial and/or temporal locality are able to serve more of the memory accesses from caches, which are much faster than main memory\(^1\).

These interactions between memory and performance have been studied in-depth in several ways. One method is to use empirical evaluation: change the software, execute on real hardware, and observe performance differences. This approach often yields good tuning and optimization results, but it is a

\(^1\)Modulo Out-of-Order execution and prefetching.
An alternative to empirical evaluation is simulation. Simulators are able to replicate both the applications’ and hardware’s behavior across varying configurations. Detailed architectural simulation [11, 27] is mainly used for evaluating architectural changes and new designs, but can also provide insight into software as it allows to reason about applications, and their sensitivities to different hardware configurations. However, it is too slow to understand applications at runtime. Recently, several works proposed to improve execution time by using high-level simulators [42, 53, 14, 47], as well as near-native execution simulation [46, 12, 40].

While architectural simulations give full insight into the behavior of the application and memory system, it is painfully slow. On the other hand, Statistical Memory Modeling has been widely used to characterize data locality by collecting architecturally-independent information. Statistical cache models [8, 23, 9, 22], for example, cheaply profile the applications and save information about their memory accesses that can be used to predict cache behavior for arbitrary cache sizes. These techniques have been successful as their implementations are fast and low overhead, while being flexible enough to model a wide range of cache sizes.

There have also been significant contributions in analytical performance models [13, 4, 26, 31]. These approaches model performance changes based on applications’ properties, and are able to predict the variation in performance (in cycles) for arbitrary architectural characteristics. This has been particularly useful to identify how applications react under different core configurations and memory hierarchy designs, in a quick and accurate way. Other examples include the analysis of sensitivity to resources such as caches [24, 38] and bandwidth [25, 37, 36].

Even though there has been extensive work looking at the connection between memory and performance, this work has generally assumed a fixed program schedule which does not hold for task-based programs, as we will illustrate in the next section.

2.2 Scheduling as a Challenge

Scheduling is determining the execution order of the tasks in the program, and it is non-deterministic: the runtime scheduler will determine the execution order according to a scheduling policy, selecting the next task from the pool of available tasks. As a result, the task schedule can change from execution to execution, unlike sequential programs.

A consequence of having different execution orders is that the applications’ overall behavior may vary, since tasks perform different operations. For exam-
Figure 2.1. Code snippet for a sample application. This structure is common in numeric solvers and image processing applications.

```cpp
A(input_data) {
    ...
    b_out = B(input_data) // B returns something based on input
    c_out = C(input_data) // C returns something based on input
    result = D(input_data, b_out, c_out) // D uses original input, plus results from B and C
}
```

Figure 2.2. Data dependency graph for the code snippet. Functions B and C are independent and use results from A. D uses output from B and C.

ple, if we examine the code shown in Figure 2.1, we will see that Function A first operates on the input and then calls two independent functions, B and C, which also work on the input data. Once both functions B and C are finished, a third function D will use the results from B and C, along with the original input data, to produce the final result. This is a common code structure for image processing applications and numeric solvers.

One way of reasoning about this structure is using a data dependency graph, as shown in Figure 2.2. The graph illustrates how A needs to be executed first, with B and C executed afterwards, which are independent from each other, and finally D combining the results. By looking at this structure we can immediately recognize the inherent parallelism of the application, and therefore, all the possible execution orders.

If we assume that the code in Figure 2.1 is a sequential application, functions A, B, C and D will be executed in that order unless the code is re-written. This can be seen in Figure 2.3 (top).

On the other hand, if the sample application is structured as tasks the execution will be different. The application will start by submitting tasks A, B, C and D to the runtime system. As the runtime sees all available tasks in the task pool, it will pick some of them for execution (e.g. task A). Once A finishes, both tasks B and C are ready to be executed, and the runtime will make a choice depending on a scheduling policy. Since the runtime follows the data dependency graph, which has multiple possible valid orderings, there can be multiple different executions without any change to the application. This vari-
Figure 2.3. Scheduling as a challenge. In sequential applications, program order is defined beforehand. In task-based applications, the execution order of the tasks is decided by the scheduler at runtime, allowing different executions for the same application. Changing the execution order affects memory behavior, which may change the overall performance.

ability changes how the application interacts with the memory system, unlike sequential programs where one sequence (ordering) of functions is followed.

When considering how tasks interact through the memory system, this scheduling flexibility during the execution becomes particularly challenging. Figure 2.3 (bottom) shows a task-based implementation of the sequential application we had before (all previous functions A, B, C and D are now tasks). The figure illustrates two different schedules for the same application, where the order of tasks B and C changes. We can see that Task D uses both the results from tasks B and C. Although B and C take roughly the same amount of time to execute, task D uses a much larger portion of data from C’s output than from B’s output.

Both schedules are logically equivalent. However, the difference in how they use data means that the schedules’ interactions through the memory system will result in significantly different performance. Let us assume 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, D will only be able to serve accesses to B’s output, because B will evict all of C’s data, making D miss on every access to C’s output. Given than D uses significantly more data from C than from B, its performance is affected by the increase in cache misses.

The increase in cache misses with Schedule 2 versus Schedule 1 is caused by the runtime scheduler’s reordering of the tasks in a memory-system oblivious manner. Schedules with increased cache misses tend to run more slowly, as illustrated in Figure 2.3.

With this example we have seen how, even if the schedules are logically equivalent, changes in the runtime schedule can result in significant performance differences due to the interactions through the memory system.

While the interactions between the scheduler and memory system can have a severe impact on performance, there is a lack of general tools to analyze and understand the reasons behind these interactions. The lack of such analysis tools limits the development of better runtime systems capable of maximizing performance under specific memory systems and platforms, by trading off some schedules for others. One example of how the runtime’s data placement can be improved by providing extra information about the application is presented in [35]. In spite of these efforts, there is still room to provide useful...
information to the runtime about the effects of scheduling in a flexible and efficient manner.

In this thesis, we address the problem of missing tools for understanding the interactions between schedules and the memory system. We propose new techniques to analyze task-based executions, applicable to both CPUs (Part I) and GPUs for graphics rendering (Part II). In the first part, we break down the different interactions of the scheduling-memory-performance triad into smaller problems, leveraging existing techniques and adapting them to the task programming model: scheduling to performance interactions (Contribution I), scheduling to memory interactions (Contribution II) and finally, the interplay between the three factors (Contribution III).

In the second part, we apply these techniques to gain insight into the world’s most popular task-based workloads: graphics rendering on GPUs. Modern video games and 3D animations deliver hundreds of frames per second, where each frame is rendered by tiling the output image and processing hundreds of thousands of tasks in parallel. This makes these applications a relevant platform for investigating the complex interactions between the tasks, the memory hierarchy and their performance. Using our contributions from Part I, we first characterize these applications according to data reuse and explore the limits of improving scheduling (Contribution IV), and finally study current data placement results, proposing solutions for this context (Contribution V).
3. Understanding Tasks Parallelism in CPUs

Previously, we discussed how the execution of task-based programs is affected by several factors. If tasks are scheduled differently, the overall performance of the applications can change due to variations in how tasks interact through the memory system. Our primary goal is to understand the role of different scheduling decisions on the impact on performance, caused by using the memory system differently. To do that we will look at specific aspects of the triad.

Figure 2.4 summarizes three new techniques we propose (Papers I, II and III), and how they cover one particular area of the Scheduling-Memory-Performance triad. First, we investigate how the performance of tasks is affected when sharing resources (interaction between scheduling and performance). Second, we explore how different schedules exhibit different data reuse (interaction between scheduling and memory, or data locality). In the end, we introduce a method to visually link changes in memory behavior caused by scheduling to the performance of the execution, connecting all three parts of the triad (Memory, Scheduling and Performance).

3.1 Contribution 1: Task Pirate

The first area we dive into is an interaction between scheduling and performance: we study how tasks behave when sharing the cache. This allows us to evaluate how sensitive the performance of the tasks is to the shared cache when co-executing.

Tasks that are running in isolation (by themselves) can fully utilize the resources such as the shared cache and busses to move data from memory. On the contrary, tasks running in parallel will fight for many of those shared resources, and particularly for the last level cache which is crucial for performance. However, the sensitivity of the tasks to sharing those resources depends on what they are computing, meaning that not all of them will be affected equally.

Figure 3.1 illustrates several examples of this. In case (a), we can see a single task in isolation, with all private and shared caches available to itself. On the other hand, a range of scenarios can occur when co-executing tasks. In case (b), tasks B and C have a similar memory behavior, which in practice translates into an even sharing of the resources (e.g. each occupying 50% of the shared cache). In case (c), task D manages to fill most of the last level cache.
Figure 3.1. Sensitivities to cache sharing. (a) Task is running in isolation, fully utilizing the cache. (b) Tasks B and C share the cache evenly. (c) Task D has a more aggressive memory access pattern, reducing the space that task B gets in the cache. (d) Task D is combined with a non-memory intensive task E, having a symbiotic relationship at the memory system level.

cache because it has a more aggressive memory access pattern compared to task B. As a consequence, task D reduces the effective space that task B gets in the cache, which might degrade the performance of B due to the increase in cache misses. Finally, case (d) shows how the tasks D and E complement their memory requirements in a symbiotic relationship, creating a nice interaction and maximizing resource usage.

All these scenarios will have different implications on the performance achieved by these tasks, and it is up to the scheduler to determine whether it is good to co-execute tasks and how. In Paper I, we leverage a method called Cache Pirating [24] to study cache sensitivity in the context of tasks.

In the original Cache Pirate work, the application subject to study is co-executed multiple times with a pirate application. In each execution, the Pirate will issue memory requests at a certain rate, and as a result, it will steal a fixed amount of the shared cache. Meanwhile, the pirate will also read and record information from the hardware performance counters of the target application to be able to compute its performance (usually cycles per instruction, and cache miss ratio).

Comparing the performance information to the cache miss ratios allows us to understand the sensitivity of the program to a particular cache pressure: the more the application uses the shared cache, the more its performance might degrade if pressure is applied with the pirate. Cache Pirating provides cache miss ratio and CPI curves for all different pressure levels. The applications that are sensitive will expose lower CPI when miss ratio is higher, whereas non-sensitive applications will show negligible difference.

The sensitivity information provided by the Cache Pirate is crucial to optimize the performance at runtime: tasks (or threads) that are sensitive should
Figure 3.2. 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.

Task Pirating

A clear advantage of Task Pirating 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, the average task behavior per type can be computed to generalize.

The Task Pirate runs with the application through the entire execution. The sensitivity data is collected at runtime, while the analysis is done offline (i.e. computing miss ratio and CPI curves). 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.

In the paper, we show how reasoning on a task-type basis allows to draw conclusions valid for many tasks. A whole set of tasks can be severely impact
when it is combined with cache-hungry tasks of a different type (e.g., tasks B and D in Figure 3.1). Similarly, some combination of tasks have a symbiotic relationship at the shared-cache, which makes them a good fit for co-executing (e.g., tasks types D and E in Figure 3.1).

This information enables the scheduler to improve scheduling decisions during runtime, since 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 (easily over 50k).

3.2 Contribution 2: StatTask

In Task Pirate, we saw how the performance of the tasks can vary due to scheduling, as co-scheduling tasks might be sensitive to cache sharing (i.e., sharing data at the same time, or spatial locality). However, there are also other reasons for why performance might vary. The example illustrated by Figure 2.3 shows how data can be reused differently over time, hurting the overall performance (i.e., using data left in the cache, or temporal locality).

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

One well-established method to study locality properties are Statistical Cache Models [8], which became widely adopted due to their speed and ability to model different cache sizes without requiring additional information.

For example, StatCache [8] (for random replacement caches) and StatStack [23] (for LRU caches) 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 allow the profiling phase to incur very low overhead, only a small fraction of the memory accesses are sampled (one every 10 thousand or more), and observed until they are reused.

After the profiling phase, the sampled data reuse information is used to model the cache behavior. First, the number of intervening memory accesses between each reuse is computed, which is often called reuse distance. With these reuse distances, a Reuse Distance Histogram (RDH) is calculated to identify how the reuses are distributed. In addition, with the RDH it is possible to determine the maximum reuse distance allowed before the data is evicted, for any given cache size. This enables modeling of different cache sizes: predicting 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, directly related to the cache size.
Figure 3.3. StatTask Problem: In task-based applications, changing the schedule (execution order of tasks) changes the way data is reused throughout the execution, and thus their reuse distances, which is a challenge for existing statistical cache models.

However, task-based applications are not a good fit for this models out-of-the-box. The executions of task-based programs change drastically based on how the runtime system schedules the tasks. If a particular schedule is profiled with StatTask, the data reuses and reuse distances identified will be tied to that schedule. A different execution order for the tasks changes their memory accesses, and therefore, the data reuses 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 3.3 shows an example of this situation. On the top, we can see a sequential application, how its memory accesses are sampled, and how StatStack identifies its reuse distances over time. On the bottom, a task-based application is shown, consisting of tasks A, B and C with two different sched-
Figure 3.4. Previous Statistical Cache Models vs. StatTask: only one profiling phase is needed instead of one for each schedule.

ules. In Schedule 1, Task B reuses data from A with a reuse distance of 5. However, in Schedule 2 task C is scheduled between A and B. Even though Task B still reuses data from A, the reuse distance in the new schedule is 15, due to the memory accesses generated by C happening between A and B.

To address this issue, some techniques were proposed [43, 34, 54] to study data reuse of task parallel runtimes. Most of them based on characterizing holistically the data locality properties of the applications. However, these techniques are not flexible enough to predict locality for arbitrary schedules or cache sizes.

In Paper II, we present StatTask, which leverages the StatCache and Stat-Stack models to be used with task-based applications. StatTask is able to 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 is focused on studying the temporal locality of the schedules (i.e., how tasks reuse data through the private caches over time). Nevertheless, it can also be used to analyze spatial locality (i.e., how tasks reuse data at the shared cache when co-running), hence, being complementary to the Task Pirate (Contribution 1).

StatTask works by profiling a single schedule of the target application. Memory accesses are collected along with new information from the task originating it, the address, the type of operation (read or write) and program counter. Later, the Reuse Distance Distribution is built for the profiled schedule, and finally, cache miss ratio curves are computed. Compared to previous models, the new information collected about the tasks allow us to classify data reuses based on the tasks involved: a reuse that only happens within one task is a private reuse, while a shared reuse happens between two tasks.

This enables one key advantage of StatTask: If the cache miss ratio curve is desired for a different schedule, StatTask can recompute the distances from the previously captured reuses by looking at the classification (private vs. shared), avoiding the need for re-profiling every possible schedule. This is illustrated in Figure 3.4: using traditional statistical cache models requires a profiling
phase for each schedule of the same application, while with StatTask only a single profiling phases is needed.

In Paper II, we study applications from the BOTS benchmarks suite, showcasing the potential of StatTask’s analysis to understand task-based scheduling. 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 new 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

In Papers I and II we presented new methods to gain insight into (1) how the performance of the tasks is affected when they are co-scheduled, and (2) how the temporal locality of the application changes due to the schedule. While these are a key step to better informed scheduling decisions and making the runtime more aware, we have not yet explored one missing link between the three factors in the triad: scheduling, memory and performance.

In Paper III, we tackle 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 which is useful to tune particular applications for specific inputs or platforms. 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 reaction from the research community, significant work on better scheduling heuristics has been proposed. For example, there are several scheduling policies, such as work stealing, that optimize for load balancing. However, they are unaware of data locality [2], which is often the main cause of achieving worse performance on memory-bound applications.

In general, developers attempt to characterize their workloads 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, simulating the tasks execution in a controlled environment [48, 16], or interactive visualization of the execution trace [28, 20, 7] to understand and tune the scheduler.
To address this gap, Paper III presents TaskInsight, a new method to characterize the scheduling process in a quantitative manner. 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 uses vital information from the data reuse between tasks for answering these questions: data reuses can be quantified over time, exposing the interactions between the tasks’ performance and their schedule. In addition, TaskInsight can interface directly with the runtime system to provide this information both to the scheduler and the programmer.

Figure 3.5 shows an overview of the TaskInsight methodology. There are two phases called **profiling** and **instrumentation**. During the first one, a profiler captures and saves information about the memory accesses of the target application 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. Read-
ing the hardware performance counters at the start and end of a task allows to study the behavior per task, but also to avoid 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.

TaskInsight requires two executions with the same schedule because the profiler is implemented using dynamic binary instrumentation, which adds an overhead to the execution and affects the native performance of the application. Reading the hardware performance counters while the profiler is attached would result in biased results. Instead, TaskInsight executes the application a second time, without the profiler attached. However, this limitation is not intrinsic to the technique: if the profiler is sufficiently low-overhead, the two steps could be combined.

Once the profiling and instrumentation phases are finished, the saved memory access are analyzed differentiating private data from shared data in a per-task basis. For any 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 task.
- **older-reuse**: If the memory address was used before, but by an older task.

In Figure 3.5 we can see this 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 provided by TaskInsight is shown in Figure 3.6, for the application histogram 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

---

¹In Paper III, the default scheduling policy is named naive. As seen in [1], 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 3.6. Example analysis of the histogram benchmark (OmpSs) with TaskInsight
information. This enables us to detect particular tasks that had a performance degradation, when where they executed, and if the reason behind the degradation is reusing old data no longer in the cache. The figure also shows how scheduling tasks that bring a substantial amount of new data in the middle of the execution (Naive schedule, task 18) can increase the number of L2 cache misses, hurting the overall performance.

In the case that multiple schedules want to be analyzed, the instrumentation phase needs to be executed for each of them. On the contrary and similarly to StatTask, the profiling phase is only needed once for any arbitrary schedule (Figure 3.7). This is one of the main strengths of TaskInsight as a low-overhead technique, since profiled data can be saved and reused to model any other schedule.

In Paper III, we study a broad range of applications. We demonstrate how TaskInsight not only shows per-task performance, but also provides an explanation for why tasks of the same type can have significant performance variation (up to 60% in our examples). 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. In addition, our findings also show how programmers can now quantitatively analyze the behavior of the scheduling policy and the runtime can use this information to dynamically make better decisions for both sequential and native multi-threaded executions.

Figure 3.7. 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.
In summary, 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. Previous approaches that only measured the actual cache miss ratios per task (e.g., using hardware performance counters) were unable to trace back changes in memory behavior and connect them to the originating scheduling decision. As a result, this novel methodology enables scheduler designers to gain insight into how specific scheduling decisions impact later tasks.
4. Understanding Task Parallelism in Graphics Rendering

In the previous chapters, we developed methods and tools to analyze the execution of task-based applications on CPUs. These methods not only leverage the information provided to the runtime systems, but also can help the developers to understand three key interrelated factors of these executions: the tasks schedule, the use of the memory system, and the application’s resulting performance.

However, the task programming model does not only apply to CPUs. Tasks have propagated to many other contexts such as Embedded Systems [19, 45, 33], HPC (e.g., through accelerators) [50, 3, 29, 18], and Graphics Rendering [44, 10, 49, 51], the latter being one of the most popular task-based workloads nowadays given the amount of mobile devices sold every year.

Modern graphics rendering is a complex multi-step process. Each frame is rendered on the GPU by merging many simpler intermediate rendering passes, called scenes. For performance, scenes are tiled into tasks that execute in parallel, producing different parts of the final output.

The diversity in what the scenes are rendering, touching different parts of the input and having different sizes cause complex memory behavior and interactions between the tasks. As a result, bandwidth demands and data sharing vary over time, and this depends heavily on the structure of the application. To design memory systems for GPUs that can efficiently accommodate and schedule these workloads, it is necessary to understand their behavior and diversity.

In Papers IV and V, we apply the models presented in the previous part to the context of graphics. In particular, we use the TaskInsight methodology to look at all tasks and all scenes across hundreds of frames from modern graphics benchmarks, in order to understand their memory behavior over time and potential optimizations.

In Paper IV, Behind the Scenes, we dive into the characteristics of a diverse set of applications ranging from complex 3D games and animations to websites. We look at their internal structure, as well as how this structure relates to their complexity at the memory system level. We apply TaskInsight to fully understand data sharing at different levels: within and between tasks, between scenes and across frames. Then, we use this information to explore the limits of scheduling by answering one fundamental research question: What would be the minimum traffic we would see to main memory (ideal case) assuming
that the scheduler does a perfect job at capturing reuses at each level (tasks, scene and frame)? Answering this question is key to determine if it is worth exploring scheduling optimizations.

While undertaking this investigation, we stumbled upon a remarkable conclusion. For these workloads, the cache sizes required to observe a meaningful difference in traffic to main memory due to scheduling are far from what is shipped today with current GPUs. This is because most of the data reuse happens across scenes, whose datasets are about one order of magnitude larger than the existing caches. On the other hand, we observed how smaller caches were largely polluted with data that was seldom, if at all, reused.

In Paper V, we built upon this observation to explore how to reduce cache pollution at the shared cache. We show how it is possible to combine high-level information provided by the graphics framework (OpenGL) with our data reuse analysis to identify data that is not worth installing in the cache. We propose a new technique that dynamically learns the amount of pollution over frames and bypasses non-reused data to reduce traffic to main memory.

4.1 Contribution 4: Behind the Scenes

Most of the resources in current GPUs (compute logic, memory bandwidth, die area, etc.) are dedicated to rendering graphics. The rendering process in modern graphics applications is a complex series of steps. Each frame being divided into several smaller rendering passes for different parts of the image and effects, called scenes. All computations in a scene are accomplished by shader programs, which consume input buffers (called resources or textures) and produces other output buffers (also, resources or textures).

In addition, these scenes are further divided into 2D tiles, which are then processed as parallel tasks across the compute resources of the graphics processor.

Figure 4.1 illustrates the complexity of the rendering process, by showing the generation of one frame from the benchmark application Manhattan. Manhattan renders a complex futuristic cityscape animation. Each frame consists of 60 different scenes, executes over 4,000 tasks, and requires more than 95MB of input data, which at 60 frames per second represents a bandwidth of 5.7GB/s.

The flow of graphics rendering is shown in Figure 4.1 (top). The application starts by rendering several scenes that store intermediate results in different output buffers (1). These buffers are combined and used as input of the following scenes (2) that add special effects, lighting (3, 4), and other details such as text and 2D overlays (5) to produce the final frame.

Figure 4.2 (bottom) shows the full complexity of rendering frame number 300 from Manhattan. Each box in the graph represents the execution of a different scene with its corresponding (partial) output images. The arrows
Figure 4.1. Rendering a frame of the application Manhattan. Schematic overview of intermediate scenes.

Figure 4.2. Rendering a frame of the application Manhattan. Details of frame 300. A sequence for the application Heaven is also shown.
Figure 4.3. Taxonomy of Memory Accesses.
between the boxes indicate how data is shared between scenes (both sharing potential and execution dependencies). Note that not all scenes use all output buffers, and that multiple input buffers may be consumed by the same scene.

The figure also highlights a fragment of the execution where multiple scenes are reading and writing to the same intermediate output buffers: Scene 1 first draws the vehicles, later Scene 2 overlays the traffic lights, and finally, Scene 48 outlines the skyline of the buildings in the background.

The combination of scenes and input/output dependencies for each frame make the rendering process very complex. Further, the actual execution is far more intricate: each scene is tiled into multiple tasks that are scheduled for parallel execution on the available hardware resources. The interaction between the scenes, the task parallelism, and scheduling leads to complex memory system behavior which is hard to understand and optimize. E.g., consumer-producer pairs of tasks may benefit from the output being in the cache, but only if it is small enough and there are few enough inputs.

Even though these complex graphics rendering workloads are ubiquitous today, previous work has not focused on understanding their behavior connected to the memory system design. In *Behind the Scenes*, we fill this gap by proposing a new way to characterize and analyze data reuse in graphics workloads, which is architecturally independent and directly linked to the application’s structure. The insights from this technique can be used to understand memory bottlenecks and potential optimizations, such as changing scheduling to maximize data reuse from the caches, or changing replacement policies to minimize cache pollution.

To accomplish this, we first present a new taxonomy to precisely describe the execution model in terms of data accesses and data reuse. Instead of classifying data into *shared* or *private* as in our work on CPUs, we define three different types of sharing:

<table>
<thead>
<tr>
<th>Inter-Frame Reuse</th>
<th>Memory addresses (data) used by multiple frames. A frame/scene uses some data if there is a task in that frame/scene that uses that data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-Scene Reuse</td>
<td>Data used by multiple scenes within the same frame, i.e. sharing between tasks across different scenes.</td>
</tr>
<tr>
<td>Inter-Task Reuse</td>
<td>Data used by multiple independent tasks within the same scene, i.e. sharing of data between tasks within the same scene.</td>
</tr>
<tr>
<td>Intra-Task Reuse</td>
<td>Data reused within a task.</td>
</tr>
</tbody>
</table>

One key property of this taxonomy is that these data-reuse relationships are a fundamental property of how the application is structured, that is, how the
frames, scenes and tasks use the data. The dynamic scheduling of tasks and the architecture does not affect how the data is shared, but it has a very significant impact on how much of this sharing can be taken advantage through the cache hierarchy.

The classification introduced by this taxonomy is illustrated in Figure 4.3. In Frame 1, we see how Scene 1 is the output of multiple parallel tasks, where some of them (top left corner) access the same data repeatedly from a texture (intra-task reuse). In the next step, Scene 2 executes. Some of its tasks (bottom left) use output data from Scene 1 (inter-scene reuse). Meanwhile, multiple tasks in Scene 2 are reading the same input texture from memory to overlay it onto the vehicles (inter-task reuse). In addition, we can see how multiple input textures and output buffers are used by Frame 2 (inter-frame reuse).

The above taxonomy allows us to identify exactly at which level the data was shared. Therefore, it makes it easier to study sharing properties inherent to the application and not to its execution. In Paper IV, we start by exploring the diversity of memory footprints across a range of applications from complex games to simple web page scrolling. This analysis reveals a wide range of memory demands across applications, within applications’ scenes and tiles, and over time, independent of the hardware and scheduling.

The conclusions from this analysis can be seen in Figure 4.4 where several applications are classified according to their scenes’ footprint diversity. On the x-axis, we the percentage (%) of scenes with datasets under a certain size threshold (y-axis). The most complex and slower benchmarks (called Low-FPS as the achieved frame rate is low) present a large diversity in the footprints of the scenes. This means that, while it is harder to cache larger scenes because of their total size, there is more opportunity to optimize the scheduling to take advantage of the data reuse between scenes. On the other hand, in fast (High-FPS) benchmarks, scenes are smaller, but there is almost no diversity in the footprints of the scenes. Thus, it is easier to cache the scenes, but there is not much opportunity to optimize scheduling if those scenes do not fit in the cache.

Figure 4.4. Footprints of scenes for different workloads, as presented in Paper IV.
After analyzing the footprints, we investigate how much data is shared at the frame, scene, and task level, as this sharing can be captured with appropriate caching and scheduling to reduce bandwidth. Scheduling strategies could take advantage of data sharing to maximize data reuse through the caches. Private caches, for example, are often able to capture many data reuses local to each task (intra-task reuses) if its data fits in the cache. In a similar way, the shared cache in a GPU can take advantage of the reuses between task or scenes if they fit.

The minimum bandwidth consumption (in MB/frame) that we would observe for an application is the frame’s footprint. However, the minimum bandwidth to main memory can only be achieved if all data reuses are captured by the caches. We build upon this observation to explore the potential of scheduling and architecture optimizations that take advantage of data reuse (both sharing and private reuse) at the scene and task levels. To understand which of the different types of sharing have the most potential for optimization (i.e. which ones affect bandwidth the most), we examine our applications running under four different scenarios:

1. **Perfect Intra-Task Sharing**: assuming perfect caching for reuses local to each task (e.g., every task fits its own data in the cache).
2. **Perfect Inter-Task Sharing**: assuming an infinite size cache for data shared between tasks of the same scene.
3. **Perfect Inter-Scene Sharing**: assuming an infinite size cache for data shared between scenes within a frame.
4. **Perfect Sharing**: Enabling all optimizations above together.

To model these scenarios, we first classify memory accesses offline and later use this classification during the execution to determine whether an access goes through the standard cache path or is logically placed in an infinite cache for the targeted shared data. The targeted shared data will always hit after its first access and does not consume space in the regular shared cache, leaving it available for whatever other data is brought in later.

By simulating these four different perfect-sharing optimizations (shown in Figure 4.5) we are able to reveal the limits of memory related optimizations and answer one fundamental question: how would bandwidth be affected by a schedule that perfectly captured intra-task reuse, inter-task reuse, and inter-scene reuse in the graphics rendering context. We show how bandwidth could be reduced by 43% on average across all workloads if we could capture reuse between tasks and scenes in a cache. For intensive, low-FPS benchmarks, bandwidth can be reduced by 40% on average from inter-task reuse (up to 76%) and 60% from inter-scene reuses (up to 75%). This new workload characterization and limit study goes beyond previous work, identifying promising directions for improving memory system efficiency in future systems.
Figure 4.5. Limit Study configurations: exploring the limits of scheduling
4.2 Contribution 5: Tail-PASS

The limit study in Paper IV revealed that most of the potential for improving scheduling of the graphics rendering relies on taking advantage of the inter-scene sharing. Although other types of sharing can also be exploited, the study shows that there is a limited opportunity for reducing traffic to main memory through improving the scheduling of tasks within a scene (i.e. optimizing inter-task sharing).

However, to be able to realize near-optimal inter-scene reuses through better scheduling, large shared caches are necessary. As a consequence, for small shared caches (such as the ones shipped today with current GPUs), it is hard to take advantage of inter-scene reuse.

In spite of this conclusion, Paper IV gives us insight in a new direction. The characterization presented allowed us to further classify the data sharing according to how often the data is accessed (reuse intensity). In addition, tasks read data from input resources and store intermediate results into output resources (also called textures). These resources can be uniquely identified with information provided by the graphics framework (such as OpenGL).

Combining both the characterization from Paper IV with the resource information provided by OpenGL, it is possible to easily determine which resources are frequently used and which ones are not. To study one particular example, we will look at one frame from the benchmark Valley (Figure 4.6). Each of the resources/textures used as input/output during the rendering of this frame has a unique ID, determined by OpenGL. Figure 4.7 shows the breakdown of memory accesses (y-axis, percentage) based on the different resources (x-axis, resource ID), sorted from most-used (left) to least used.

The height of each bar shows the accesses (percentage) to each resource, and the colors indicate the miss ratio of the scenes accessing that data. Red indicates the portion of accesses from scenes with miss ratios over 80%, while green below 80%.

It is interesting to observe how most of the resources in the tail of the distribution (tail-resources) expose very low reuse: the data from those resources will not benefit from being installed in the cache. On the other hand, more than 80% of the data accesses are to the top 10 most accessed resources. Even though there are over 600 different resources being used in total, in the event of a cache miss, the likelihood that the miss will come from an access to one of the top 10 resources is more than 80%.

In spite of this observation, most of the current shared caches have an always-insertion policy, which means that all the data is cached regardless of how much it is reused. A large number of not-reused resources (such as in the case of Valley) can cause pollution in the cache that evicts reused data, increasing the traffic to main memory.

To study this effect, in Paper V we perform a simple experiment: only allow the top-10 resources to be cached, bypassing the tail-resources. Figure 4.8
Figure 4.6. One frame of the application Valley from Unigine. Each frame goes through over 80 different scenes, 21,000 tasks, and uses over 600 different resources.

Figure 4.7. Breakdown of cache accesses per resource, colored by the scenes miss ratio: red is accesses from scenes with miss ratios to those resources over 80%. Green is under 80%
shows the results from running this experiment on a sequence of 20 frames of Valley. The non-bypassed execution (baseline) is shown on the left half of the figure, while the right side shows the results when bypassing the tail-resources.

On the left, the distribution of the accesses based on the resource-IDs is shown. Approximately 40% of the total accesses come from resource number 390, 10% from resource 389, 8% from resources 260, etc. On the other hand, accesses to the tail-resources represent 18% of the total. However, those accesses are being installed in the cache but missing the most (last bar tail in Figure 4.8(a)).

Figures 4.8(b) and (c) show the number of shared L2 cache misses coming from each resource. Most of the misses are coming from the tail-resources, and most of those resources are already performing poorly with miss ratios over 80%. This suggests that if we do not cache these resources, the application will suffer very little, while making more space available for well-performing resources.

If we look at the results of the execution when bypassing the tail-resources (c), we observe the following: first, there is an extra penalty for bypassing the tail. We can see how there is an increase of 10M (15%) in the number misses to tail-resources due to the fact that they are never cached (shown in red). Second, and more significantly, there is a benefit on all other resources, where the misses are dramatically reduced as a result of avoiding the cache pollution from the tail.

This effect can be seen in Resource 390, which is the most accessed one. In the bypassed execution, it exhibits 5M (13%) more hits than before. In addition, other resources such as 257, 258 and 259 range from 9M to 17M (64% to 75%) more hits.

As a result of bypassing the least frequently accessed, and least cache-friendly tail-resources, we see a decrease in cache pollution leading to a 20% reduction in the misses to the L2 shared cache. Accesses to the top resources that were missing before can now be served from the cache (shown under overall misses in the figure). While the tail-resources do incur more misses, as long as the benefit in the top-10 resources exceeds the penalty to the tail resources, the overall number of misses will be reduced.

Repeating this experiment of simply bypassing the tail-resources for 50-100 frames of the benchmarks in Paper IV results in 7% fewer cache misses on average, and up to 35% improvement in some frames/scenes.

This experiment gives two key insights into the high-level structure of graphics applications: First, the most accessed (top) resources make up the vast majority of the reused data. Second, the least accessed (tail) resources have much higher miss ratios. Combined, this represents an opportunity to improve cache behavior by bypassing the tail.

With these results, in Paper V, we present an approach called Tail-PASS, that detects the top-n resources and dynamically enables or disables bypassing of the tail-resources.
Figure 4.8. Motivation Example: Potential of bypassing.
Figure 4.9. Implementation. Tail-PASS learns in 2 frames if it is worth bypassing the tail to optimize future frames.

Figure 4.9 shows an overview how Tail-PASS works. First, a learning period of two consecutive frames identifies the top-N resources and evaluates the benefit of enabling tail bypass. During the learning period, the first frame is executed with no cache bypassing (NOBYP) and the second with the tail bypassed (BYP). In the case shown in Figure 4.9, frame 1 renders in non-bypassed mode (purple). The frame consists of 60 scenes, and for each of them the total number of cache misses is recorded, and a table of the top-N resources is built.

The next frame is executed in bypassed mode, using the top-N resource list built in the first frame to determine caching and bypassing. During this frame, the total cache misses per scene are also recorded in a per-scene basis.

Once the learning period is finished, Tail-PASS does a scene-by-scene comparison of the aggregate miss ratios to determine if the benefits of bypassing outweigh the penalty for increased tail misses, and applies this to future scenes. In Figure 4.9 we can see how Scene 1 had fewer misses when executed non-bypassed. Based on this, bypassing will be disabled for future executions of Scene 1. Similarly, Scene 2 performed better when bypassing the tail-resources, and therefore, subsequent executions of it will enable bypassing.

The Tail-PASS methodology is built-upon the results presented in [5], where it is shown that consecutive frames share over 98% of the texture (resource) accesses. This conclusion allow us to quickly learn which approach works best in the learning period, and dynamically apply this insight for future frames with high accuracy, since it is extremely likely that the coming frame will share most resource accesses. To adapt to changes in frames over time, the learning period is repeated every certain number of frames.

Tail-PASS uses high-level information provided by OpenGL (scenes and resources) to follow the application’s phases and behavior, and to select datasets that are infrequently used to give more space in the cache to frequently accessed data.
Applying Tail-PASS to the same workloads as in Paper IV reduces bandwidth consumption by 22% on average (up to 46%) in high-end graphics applications if top-10 resources are tracked, and by 11% on average (up to 44%) for just the top-2 resources.

Tail-PASS lays the foundation for improving state-of-the-art cache management, using information already given by modern graphics frameworks, such as OpenGL, to utilize the memory system more efficiently.
5. Conclusion

While the hardware on modern CPUs and GPUs is becoming more complex, dozens of cores and deep memory hierarchies, task-based programming stands out as a simple and flexible option for parallel programming. Since it allows to delegate several intricate decisions to a runtime system, such as task scheduling and data placement, it recently gained popularity among developers.

In order to develop better runtime systems and optimize their performance, it is crucial to understand three key components: scheduling, data locality (memory behavior) and performance, and how they affect each other. Current approaches give an accurate view of the application on one particular system, but they are oblivious to executing with different schedules. As a consequence, 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 models and techniques 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 how the performance of the application was affected by changes in the memory behavior of the tasks because of scheduling (Paper III). 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.

In addition, we demonstrate the use of these techniques by applying them to understand the world’s most popular task-based workloads today: modern graphics rendering. We analyzed how graphics tasks interact with the memory system and explored the potential benefits for doing better scheduling (Paper IV), and finally, we propose a new cache-management approach that is based on the analysis of data reuse of the graphics tasks, and that uses high-level information from the application’s structure provided by the graphics programming framework (Paper V).

These methods lay the foundation for leveraging task-based runtime systems to fully understand the applications, and utilize memory systems more efficiently.
6. This Thesis in a Nutshell

<table>
<thead>
<tr>
<th>Problem</th>
<th>What is the main problem?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap</td>
<td>Why is current work not sufficient?</td>
</tr>
<tr>
<td>Solution</td>
<td>What is the main contribution?</td>
</tr>
<tr>
<td>Results</td>
<td>How did things change with our contribution?</td>
</tr>
</tbody>
</table>

**Paper I: Task Pirate**

<table>
<thead>
<tr>
<th>Problem</th>
<th>The performance of tasks that are co-executed can be affected due to cache sharing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap</td>
<td>Existing work studied cache sharing at the <em>entire application</em> level, but not at the <em>individual tasks</em> level.</td>
</tr>
<tr>
<td>Solution</td>
<td><strong>Task Pirate</strong>: intentionally steal cache from tasks to study their sensitivity to cache-sharing.</td>
</tr>
<tr>
<td>Results</td>
<td>There is significant diversity in the sensitivities, but limited options in the pool of available tasks to improve scheduling.</td>
</tr>
</tbody>
</table>

**Paper II: StatTask**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Changes in the execution order of the tasks (schedule) impacts how data is reused over time, and hence, what is served from the caches.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap</td>
<td>Existing work studied reuse behavior for a <em>fixed</em> schedule. If previous work is used for task-based applications, the conclusions are scheduled-dependent.</td>
</tr>
<tr>
<td>Solution</td>
<td><strong>StatTask</strong>: formalize reuse distances in a task-based context, and model schedule changes to predict how memory behaviour changes.</td>
</tr>
<tr>
<td>Results</td>
<td>Our approach is able to predict results for arbitrary schedules from a single profiling run.</td>
</tr>
</tbody>
</table>
### Paper III: TaskInsight

<table>
<thead>
<tr>
<th>Problem</th>
<th>The memory behavior of tasks changes according to the schedule, how does this impact the performance?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap</td>
<td>Existing work studied either performance changes or memory behavior changes, but cannot detect points in time were bad scheduling decisions were made and explain why.</td>
</tr>
<tr>
<td>Solution</td>
<td><strong>TaskInsight</strong>: analyzes reuse behavior, measures performance changes and connects all this information displaying it over time, to visually understand the impact on performance.</td>
</tr>
<tr>
<td>Results</td>
<td>It is possible to identify bad scheduling decisions, measure their impact, and understand the underlying reason behind it.</td>
</tr>
</tbody>
</table>

### Paper IV: Behind the Scenes

<table>
<thead>
<tr>
<th>Problem</th>
<th>Graphics workloads are the world’s most popular task-based workload today. The interactions of the tasks at the memory-system level is very complex and it is hard to find optimizations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap</td>
<td>There is no systematic approach to understand how graphics workloads use current memory systems in GPUs and find optimizations to reduce traffic to main memory.</td>
</tr>
<tr>
<td>Solution</td>
<td><strong>Behind the Scenes</strong>: new taxonomy to characterize the data reuse and sharing of the rendering tasks in an architecturally-independent manner. In addition, a limit study to understand the potential for doing better scheduling.</td>
</tr>
<tr>
<td>Results</td>
<td>There is a significant diversity in the footprints of the graphics tasks, and in the amount and types of data sharing when rendering graphics. This leads to complex memory behavior that we can now understand with a new taxonomy. Limit study reveals significant potential for doing better scheduling across scenes, and to reduce cache pollution in small caches.</td>
</tr>
</tbody>
</table>
## Paper V: Tail-PASS

<table>
<thead>
<tr>
<th>Problem</th>
<th>Shared caches in GPUs use always-insertion policies. Rendering tasks tend to use significant data that is never reused, generating a lot of pollution, which increases traffic to main memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap</td>
<td>Existing work does not utilize high-level information from the application’s structure provided by the graphics programming framework to optimize data placement at the shared cache.</td>
</tr>
<tr>
<td>Solution</td>
<td><strong>Tail-PASS</strong>: dynamically detect and bypass non-frequently-reused resources, using information provided by OpenGL.</td>
</tr>
<tr>
<td>Results</td>
<td>Significant reduction in traffic to main memory for small cache sizes.</td>
</tr>
</tbody>
</table>
7. Svensk Sammanfattning

Att maximera prestandan för datorsystem och samtidigt öka deras energieffektivitet är vitalt för framtida utveckling i ingenjörskap, medicin, underhållning, med mera. Den ökande komplexiteten i mjukvara, hårdvara, och interaktionen däremellan gör dock denna uppgift svår. Mjukvaruutvecklare måste hantera komplexa minnesarkitekter säsom flernivå-cacher i moderna CPUer, och hålla tusentals kärnor upptagna i GPUer, vilket försvårar programmeringsprocessen.

Taskbaserad programmering tillhandahåller abstraktioner på en hög nivå för att förenkla utvecklingsprocessen. I denna modell skickas oberoende uppgifter (tasks) till en exekveringsmiljö som arrangerar deras körning på hårdvaruresurser. Denna angreppsvinkel har blivit populär och framgångsrik eftersom exekveringsmiljön kan distribuera arbeten över hårdvaruresurser automatiskt, och för att den har potentialen att optimera körningen med avseende på att minimera dataförflyttningar (till exempel genom att vara medveten om cachehiearkin).

För att bygga bättre exekveringsmiljöer behöver vi nu förstå prestandaflakshalsar för samtida och framtida flerkärniga arkitekter. Oturligt nog, eftersom majoriteten av dagens forskningsresultat är designat för sekventiella eller trådbaserade arbeten så finns det en generell brist av verktyg och metoder för att få insikter om exekveringen av dessa program som skulle kunna ge både exekveringsmiljön och programmerare möjligheter att hitta potentiella optimeringar.

I denna avhandling tar vi oss an denna brist på verktyg genom att tillhadahålla snabba, noggranna, och matematiskt korrepta modeller för att förstå exekvering av taskbaserade program. Vi fokuserar dessa modeller kring tre nyckelaspekter av körningen: minnesbeteende (datalokalitet), schemaläggning, och prestanda. Våra bidrag ger insikter om samspellet mellan schemats beteende, återanvändning av data genom cachehiearkin, och den resulterande prestandan. Dessa bidrag lägger grundplåten för att förbättra exekveringsmiljöer.

Vi applicerar först dessa metoder för att analysera en varierad uppsättning CPU applikationer, och utnyttjar dem sedan för en av de mest vanliga arbeten i dagens system: grafikrendering på GPUer.
Maximizar el rendimiento de las computadoras y hacerlas más eficientes energéticamente es vital para futuros desarrollos en ingeniería, medicina, entretenimiento, etc. Sin embargo, la creciente complejidad del software, el hardware y sus interacciones dificultan esta tarea. Los desarrolladores de software deben lidiar con arquitecturas de memoria complejas como los cachés multinivel en las CPU modernas y mantener miles de núcleos ocupados en las GPU, lo que hace el proceso de programación más engorroso.

La programación basada en tareas proporciona abstracciones de alto nivel para simplificar el proceso de desarrollo. En este modelo, tareas independientes (funciones) se envían a un sistema runtime, que organiza su ejecución a través de los recursos de hardware. Este enfoque se ha vuelto popular dado que el sistema runtime puede distribuir la carga de trabajo a los recursos de hardware automáticamente y tiene el potencial de optimizar la ejecución para minimizar el movimiento de datos (por ejemplo, conociendo la organización del sistema de memoria).

Sin embargo, para crear mejores sistemas runtime, necesitamos entender los factores limitantes del rendimiento de las arquitecturas multicore tanto actuales como futuras. Desafortunadamente, dado que la mayoría de los avances actuales se diseñaron para aplicaciones secuenciales o basadas en hilos, existe una falta general de herramientas y métodos para obtener información sobre la ejecución de estas aplicaciones. Esto dificulta que el sistema runtime o los programadores detecten posibles optimizaciones.

En esta tesis, cubrimos esta falta de herramientas al proporcionar modelos rápidos, precisos y matemáticamente sólidos para comprender la ejecución de aplicaciones basadas en tareas. En particular, centramos estos modelos en tres aspectos clave de la ejecución: el comportamiento de la memoria (localidad de datos), la planificación de las tareas (scheduling) y el rendimiento. Nuestras contribuciones proporcionan información sobre la interacción entre la planificación, la reutilización de datos a través de la jerarquía de caché y el rendimiento obtenido por la aplicación. Estas contribuciones sientan las bases para mejorar los sistemas runtime. Primero aplicamos estos métodos para analizar diversas aplicaciones en CPU y luego aprovechamos las mismas técnicas para estudiar una de las áreas de uso más comunes en los sistemas actuales: la renderización de gráficos en GPUs.
9. Acknowledgements

Not many times in life does one have the chance to work with such amazing and talented people. I was blessed in this regard, and would like to take a moment to thank all those who made this thesis possible.

I have to start by thanking my awesome supervisors, David Black-Schaffer and Erik Hagersten. Their guidance and support are out-of-this-world. The amount of time they spent working on these projects with me made this journey a life-changing experience. I still struggle to find the right words to thank you for the opportunity to join the UART group. I would also like to thank Stefanos Kaxiras and Magnus Själander for their support, for all the insightful discussions, and their help in reviewing. Working with such giants was both inspiring and enlightening.

I would also like to thank my co-authors. They turned the research process into a fun experience and helped me understand the challenging research questions we investigated. Andra-Hugo and Thomas Grass: thanks for all the discussions about tasks and data sharing, and for all the time we spent together. Andreas Sembrant and Trevor Carlson: thanks for letting me work in your team and for improving my critical thinking. Andreas is the first Swedish friend I made when I first came to Uppsala. Since then we have shared so many crazy experiences together, including trips to Macchu Picchu (Perú), Austin and San Francisco. I am looking forward to many more of those.

The UART group creates amazing, everlasting relationships between people. I finish my Ph.D. with many new friends for life, and I want to thank them too. This, for sure, is the part I will miss the most. First, thanks to my incredible office-mates, Ricardo Alves, Moncef Mechri, Gustaf Borgström, and Hassan Muhammad. I cannot think of a better, funnier office in the entire Polacksbacken. Ricardo and Moncef: at some point, we all were the newbies, and having you by my side was supportive and fun, making every new challenge seem like a cakewalk. Gustaf, thanks for making me smile every day, looking at those erotic plots while we brewed kaffe.

But wait: Mehdi Alipour, Johan Janzén, Kim Anh-Tran, Chris Sakalis, Marina Shimchenko, Stephan Brandauer: your offices are pretty cool too :). I want to thank all of you for the great time during fika. Mehdi, thanks for the countless Rullan lunches and conversations about life. Johan, thanks for the unlimited help you have given to me, including the translation for the Swedish Summary (Svensk Sammanfattning) in this thesis. Skydiving together is still on my TODO list. Kim, thanks for always bringing new and refreshing energy no matter how the week was going. Chris, you know so much about everything.
that I am always looking forward to our next conversation. Thanks for all your help, and for teaching AVDARK together. We will miss you next year when Johan and I will not be there. In the past year, I got a bonus chance to work with two amazing people: Gregory Vaumourin and Mihail Popov. Thanks for being such kind and friendly colleagues, and for taking the time to hear my presentations and read the papers in this thesis. Mihail, please make sure that the calendar on my desk gets updated.

I want to add special thanks to Alexandra Jimborean, Alberto Ros, and Linnea :), for the outstanding support since day one. Alexandra and Alberto always treated me like family, and now we are.

Some of the incredible people I met here have since graduated. They inspired me throughout the years, and helped me a lot at the beginning of my Ph.D. I want to thank Nikos Nikoleris, Andreas Sandberg, Vasilios Spiliopoulos, Xiaoyue Pan, Muneeb Khan, and Konstantinos Koukos. All of them have taken different paths, but each time we meet feels like if nothing has changed.

In addition, I made excellent friends in the IT Department. Rubén Cubo was the first one since we both came to Uppsala at the same time and lived together for a year. Thanks, Rubén for all the time we shared and for making my first year feel like home. Mikael Laaksoharju and Virginia Grande Castro, I was sitting in your office in my first year and it was a real pleasure. Still today I remember those days with much joy. Anna-Lena Forsberg and Ulrika Andersson, you have helped so much, even before I came here, that none of this would have been possible without you. Loreto Skoknic and Ulrika Jaresund, thanks for the countless hugs every fika break, which made me keep going for the rest of the day.

There are other people outside of Uppsala that made this possible. Thanks to Fei Wu and Adam Leffert, for our magical connection, for pushing me forward and for making me explore new grounds. Thanks to ALL my friends from Córdoba and my hometown, Ushuaia, who were always in touch and motivating me to keep working hard. I also would like to dedicate part of this thesis to all the people I lost during this journey, including my grandparents and other relatives. Thanks to all my family, including Sonia, Valeria, Eliana, Maricel, Mauricio, Rubén, Chicho, and my little cousins.

Finally, I want to close with two very special mentions. First, thanks to my parents, Verónica and René, and my brother Pablo. Last, but certainly not least, thanks to the love of my life and partner, who was there in every situation to back me up, trusting me and helping me in this crazy journey. I love all of them wholeheartedly.

To all, I will never forget all your help and support, and I hope one day I can return it. No. Better yet, I will work every day to return it.
This work was supported in part by the Swedish Research Council through UPMARC (Uppsala Programming for Multicore Architecture Research Center), by the Swedish Foundation for Strategic Research (SSF) [project FFL12-0051], and by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme [grant agreement No 715283]. Part of the work in this thesis 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].

Simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at UPPMAX (Uppsala Multidisciplinary Center for Advanced Computational Science) at Uppsala University, and at NSC (The National Supercomputer Centre in Sweden), at Linköping University.
References


[10] Tim Biedert, Kilian Werner, Bernd Hentschel, and Christoph Garth. A Task-Based Parallel Rendering Component For Large-Scale Visualiza-


[34] Hao Luo, Xiaoya Xiang, and Chen Ding. Characterizing Active Data Sharing in Threaded Applications Using Shared Footprint.


Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm.


A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)