SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Low Latency Scheduling on Many-Core CPUs

Ismail Safa Toy

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Low Latency Scheduling on Many-Core CPUs

Niedriglatenz-Scheduling auf Mehrkern-CPUs

Author:	Ismail Safa Toy
Supervisor:	Prof. Viktor Leis
Advisor:	Marcus Müller
Submission Date:	Submission date

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, Submission date Ismail Safa Toy

Acknowledgments

Abstract

This thesis improves parallel query execution by enhancing a scheduler based on the morsel-driven framework to effectively utilize the capabilities of modern many-core processors, with a specific focus on NUMA (Non-Uniform Memory Access) challenges. The growing decentralization of memory controllers in many-core systems complicates efficient data access and task scheduling, necessitating new approaches to maintain high performance and scalability. The enhanced scheduler dynamically assigns small data fragments, or morsels, to worker threads, adjusting in real-time to changes in data access speeds and task demands. This approach directly addresses NUMA-related issues by optimizing data locality and minimizing unnecessary data movement across different memory nodes, thus achieving elasticity. Extensive benchmarks, including comparisons with other multithreading schedulers like oneAPI and OpenMP, demonstrate the scheduler's effectiveness, especially in small and big tasks.

Contents

1 Introduction

1.1 Motivation

In the evolution of computer architectures, several developmental phases have brought forth different problems and their respective solutions. From unified memory to Non-uniform memory access (NUMA) bringing cache hierarchies, from single core to many-core designs necessitating robust scheduling, modern CPUs have become more complex with each iteration. The latest trend is to increase computing power with more cores and more cache per core [**modernCPUs**].

Even before the emergence of the first multi-core CPU, scheduling with multithreading in mind was heavily researched. Initially, scheduling was designed for compute clusters with more than one CPU core or for multiprogramming on a single core. Nowadays, a commercial CPU has 8-16 threads, which shifts the focus from making multithreading and multiprogramming possible to making them efficient and fast [**multithreadingEvolution**].

Parallel programming is necessary in today's computer architectures to take full advantage of their capabilities. In this context, many programming tools have been developed: from libraries such as Java concurrency API [**JavaConcurrency**] and Intel's OneTBB [**IntelTBB**], to standalone programming languages like NVIDIA CUDA [**CUDA**] and X10 [**X10**], or language extensions such as OpenMP [**OpenMP**] and MPI [**MPI**] for automatic parallelization.

1.2 Problem Statement

At the heart of parallel programming lies the division of a program's execution into so-called tasks, which are dispatched to threads to be executed concurrently. The aforementioned tools give developers the ability to specify strategies for partitioning execution. Some also support different scheduling and dispatching schemes.

However, as we move into the era of many-core CPUs with tens or hundreds of cores on a single chip, traditional scheduling approaches face significant challenges. The increasing core count exacerbates issues of load balancing, cache coherence, and memory access latency. In particular, low latency scheduling becomes crucial for a wide

range of applications, from real-time systems and high-frequency trading to interactive user interfaces and responsive cloud services [**lowLatencyImportance**].

From a computational intensity perspective, logical division of the execution is relatively inexpensive. The real challenge lies in the orchestration of concurrent execution with minimal latency. Fully utilizing cores while maintaining low latency requires sophisticated load balancing, which in the literature is achieved either by Work Sharing, where a thread with work deliberately gives up some work, or Work Stealing, where threads without tasks steal from threads with work [**loadBalancing**].

1.3 Significance of the Work

The need for low latency scheduling on many-core CPUs is becoming increasingly critical in various domains:

- **Real-time Systems:** In applications such as autonomous vehicles, industrial automation, and financial trading systems, even millisecond delays can have severe consequences [**realTimeSystems**].
- **Cloud Computing:** With the growing adoption of microservices architectures, efficient scheduling is crucial for maintaining responsiveness and managing resources effectively in large-scale distributed systems [**cloudComputing**].
- **Scientific Computing:** Many scientific simulations and data analysis tasks require both high throughput and low latency to process vast amounts of data in reasonable timeframes [**scientificComputing**].
- **Interactive Applications:** As users expect near-instantaneous responses from applications, low latency becomes a key factor in user experience and product success [**interactiveApps**].

By addressing the challenges of low latency scheduling on many-core CPUs, this work aims to contribute to the broader goal of enabling more efficient, responsive, and scalable computing systems across these critical domains.

1.4 Work Stealing and Its Challenges

Work Stealing is a provably efficient scheduler of parallel computations: the expected runtime of Work Stealing with total instructions W, critical path length S on P processors is $O(W/P + S)$, which is a constant factor away from optimal scheduling [**WS**]. Work Sharing has a worse worst-case performance and a similar average case.

The performance of work stealing is not only theoretical; it is also the standard mode of scheduling used by many programming tools. While the logic of work stealing remains simple, there are variations to the data structures which hold the tasks per thread. Concurrent deques are used because of their $O(1)$ time complexity. However, synchronization of the deque, such as memory fence or compare-and-swap operations, is a major contributor to the runtime; even a single memory fence can account for up to 40% of the total runtime [**syncronization**].

1.5 Contributions

This work aims to address some inefficiencies of Work Stealing in the context of many-core CPUs. Our key contributions are:

1. A cache-aware partitioning scheme: Inspired by "Morsel driven parallelism" [**morsels**], our partitioner uses morsels to enforce cache locality of the whole range (part of the execution). This greatly improves cache performance without noticeable overhead.
2. A hybrid scheduling mode: We introduce a hybrid mode of scheduling incorporating both Work Sharing and Work Stealing. By utilizing Mailboxing, work is delivered to threads before stealing is allowed, significantly improving task start times without causing substantial overhead in total runtime.
3. Deque optimizations: We experimented with various types of deque optimizations, such as Private, Half-Private, and Chase-Lev, as well as share-half methods. This resulted in a scalable, cache-aware scheduler with provable efficiency and low latency.

These contributions address key challenges in low-latency scheduling for many-core CPUs:

- Broken Cache Locality
- Start and inter-task latency
- Overhead related to concurrent deques

1.6 Thesis Outline

The rest of this work is structured as follows: Chapter 2 discusses the background of this topic in more detail. Chapter 3 introduces the techniques used in our scheduler.

Chapter 4 describes the concrete implementation. Chapter 5 evaluates the performance of our scheduler, and lastly, Chapter 6 discusses our work, its limitations, and future directions.

2 Background and Related Work

The Background section of this thesis establishes the foundation for understanding a schedulers principles, technologies, and practical considerations.

2.1 Evolution of CPU Architectures

In the last decades, computer performance has become faster with each generation. This trend described by Moore's Law, which predicts that the number of transistors in a dense integrated circuit doubles about every two years. While not a natural law, this industry trend has been observed from the 1970s to the present day, driving significant advancements in computing power and efficiency[**Moore2006**].

Initially, lithographical improvements enabled more transistors in a smaller package, directly translating to faster processors. For example, the manufacturing process improved from 180nm (Pentium III in 2000) to 32nm (Westmere in 2010) [**Bohr2011**]. During this period, clock speeds increased from around 1 GHz to almost 4 GHz. It is also important to note that the effective increase in clock speed is higher due to Instructions per Cycle improvements.

Unfortunately the lithographical improvements were not sustainable, due to the physical properties and manufacturing difficulties in smaller manufacturing processes. As a result the rate of improvement slowed down, the focus shifted to increasing clock speeds and improving IPC. However, this approach faced also physical limitations, primarily due to power consumption and heat generation [**Borkar2007**].

Around 2005, a paradigm shift occurred in CPU design: instead of pushing for higher clock speeds, manufacturers began to increase the number of cores on a single chip. This marked the beginning of the multi-core era, with dual-core processors becoming mainstream and quad-core processors following shortly after [**Geer2005**].

CPUs not only developed into multi-core, they were able to do simultaneous multi-threading (SMT or also Hyper-Threading). SMT enabled more computing power in the same die. Becuase a single core can execute two threads simultaneously, cpus become more forgiving towards less optimal memory usage. As waiting for memory did not stall the entire core, when a thread is expecting memory. This flexibility allowed easier multiprogramming in the early 2000s.

Moore's Law continued to be true, but the contributers to the increase in transistors changed. We observe a logarithmic development in clock speeds, power consumption and performance per clock. This means improvements are in logarithmic relation, which is not appropiate for linear increase in Moore's Law.

After a short stagnation, mostly due to lack of competition, number of cores are increasing steadily. Leading to the development of many-core CPUs, particularly for enterprise use cases. These processors boast significantly more cores than consumer processors, with some modern server CPUs featuring 64 or even 128 cores [**AMD2021**]. However, this shift introduced new challenges, often referred to as the "multicore crisis" [**Sutter2005**]. Adding more cores to a CPU often necessitated a decrease in clock speeds due to power and thermal constraints. Even today's many-core CPUs typically have clock speeds about 1-2 GHz lower than their consumer counterparts. Although, many-core cpus have more sophisticated memory arrangements, often having a decade lead in terms of bandwidth, latency and error correction. For example, latest server cpus have often 8 or 12 channel memory, Error Correction Code (ECC) memory support as well as higher speed interconnects. These advantages allow to saturate the cores.

The evolution towards multi-core and many-core architectures brought broader memory bandwidth and more cache per core. Still, memory outside of the processor needs to be accessed via the network connection, this operation stalls the core. Worse yet, the CPU operates considerable faster than the main memory it uses. Cpus found themselves increasingly starved for data, also called Von Neumann Bottleneck. As the portion of memory fethcing in the program execution increases, runtime becomes longer, this is especially true with multithreading. To minimize the time spent waiting, modern CPUs often feature complex cache hierarchies with multiple levels (l1, l2, l3), each with different sizes and access times [**hennessy2019**].

Cache performance has become crucial for overall system performance. Memory-intensive applications with suboptimal cache usage are slower, even with multithreading, than implementations with better cache utilization. This is because main memory access is prohibitively expensive compared to cache access, often by an order of magnitude or more [**Drepper2007**]. Data locality, or cache locality, needs to be cared for for optimal performance.

To mitigate this problem, modern processors employ techniques such as prefetching to minimize cache misses. Additionally, cache-oblivious algorithms and data structures can outperform those with better asymptotic runtime but worse cache performance [**Frigo1999**].

Cache	Alder Lake P core	Alder Lake E core	Zen 4
Level 1 code	32 kB	64 kB	32 kB, latency 4 clocks
Level 1 data	48 kB, latency 5	32 kB, latency 3	32 kB, latency 4 clocks
Level 2	1280 kB, latency 15	2048 k, latency 20	1 MB, latency 14 clocks
Level 3	4–30 MB, latency 65, shared	4–30 MB, shared	32–64 MB one per 8 cores, latency 47

Table 2.1: Cache sizes and latencies for Alder Lake (P and E cores) and Zen 4 architectures

2.1.1 Cache Coherent Non-Uniform Memory Access (NUMA)

As core counts increased, traditional symmetric multiprocessing (SMP) architectures faced scalability challenges. This led to the development of Non-Uniform Memory Access (NUMA) architectures, where memory access time depends on the memory location relative to the processor [**Lameter2013**]. NUMA introduces additional complexity to scheduling and memory management, particularly in many-core systems. As shown above, utilizing the cache optimally is key to not incur unncessary memory fetch from the main memory, which can be around 150-200 cycles and even longer if the access was to remote memory. Waiting for data is especially crucial in parallel programming, as waiting or blocking of the execution impacts the whole execution time. Often a simple inefficiency snowballs into a considerable chunk of wasted execution time. NUMA enhances memory performance in two ways. Firstly, it reduces the memory latency for recently used data. More importantly, it reduces the number of accesses to the main memory. Therefore NUMA is beneficial for workloads with high memory locality of reference and low/no lock contention. A hardware design challenge for NUMA architectures is to preserve coherence between caches of different NUMA clusters. A coherence problem stems from the tiered structure of the Cache itself. Up until Sandy Bridge EP and Bulldozer architectures the Last Level Cache (LLC) was not tiered and shared among all cores. As private cache (L1 and L2) per core emerged, the coherency between the caches needed to be kept. The usual memory coherence protocol MESI can be used, though at this level the communication for the Shared (S) state would be too high on the system. In a cpu with 64 cores with their respective caches, a read request can in the worst case trigger many cache snoops and their answers from the caches (cachelines) in state S. Modern cache architectures apply the MESIF protocol introducing Forwarding (F) state. One cacheline is promoted to the F state, so it is the only cache that can respond to requests. The F state indicates "first among equals". In the MESIF protocol it reduces the traffic on the rings or on the interconnects. [**Goodman**]

Current trends are towards heterogeneous unified memory access (e.g. CPU + GPU)

and towards coherent addressing of remote memory. With more compplex memory controllers, a cpu can directly address the memory of an accelerator. Modern designs often incorporate multiple levels of memory. AMD's infinity fabric allows tighter integration of shared memory among its devices (both CPU and GPU), while NVIDIA's CUDA and Intel's OneAPI are developed to give better access to their respective platforms. The goal is to reduce memory related bottlenecks. A cpu needs to request the execution data and communicate with the accelerator, and that accelerator needs to load the memory as well. In the hUMA architectures a cpu can directly load the data related to the execution on the accelerators memory.

2.1.2 Optimizations

waiting for data is often the main cause of low performance. To mitigate this problem, modern CPUs employ a plothera of solutions to minimize this time. Most important optimization is prefetching. Prefetching, loads a cacheline (mostly 64 Bytes) into the cache before the region it was loaded from, is requested. This is adventageous as it can be done parallel to another execution and it is computationally cheap. Modern CPUs have built in prefetching. Hardware prefetching is more efficient than explicit software prefetching in most of the cases.[**agnerfog**]

Out of Order Execution (OOE) is another way to reduce the time spent waiting, by reordering the execution so that independent instructions can be executed while waiting for memory to be fetched. OOE is more suitable for unpredictable access patterns, as prefetching would miss the next access, thus causing overhead. For compute bound workloads OOE is more impactful, where prefetching shines at memory bound workloads.[**OOE**]

Branch Prediction and Spekulative Execution are both probabilistic methods to select a branch in the execution that is likely to be the correct path in the conditional. The CPU continues to execute and/or prefetch the instructions and data in that path. If the executed path is the wrong path, then the executed pipelines are discarded. This is also called a branch misprediction. The maximum wasted cycles are the length of a pipeline. A branch miss is more likely if the conditional is close to random. If the conditionals result is distributed according to some distribution, the branch prediction is more likely to be correct.[**agnerfog**]

2.1.3 Implications for Parallel Programming

The shift to many-core architectures has profound implications for software development and system optimization. The real improvement in computing power now lies less in waiting for faster hardware and more in writing cache-aware, scalable parallel pro-

grams that can efficiently utilize anywhere from 16 to thousands of cores [**Keckler2011**]. This evolution underscores the critical importance of effective scheduling algorithms, particularly those that can provide low latency in many-core environments. Schedulers must now contend with:

1. Increased core counts and potential for parallelism
2. Complex cache hierarchies and the need for cache-aware scheduling
3. NUMA effects and the importance of memory locality
4. The need to balance high throughput with low latency for diverse workload

In the context of this thesis, understanding these architectural trends is crucial for developing effective low latency scheduling algorithms for many-core CPUs. The challenges posed by increased core counts, complex memory hierarchies, and diverse application requirements form the foundation for our research into advanced scheduling techniques.

2.2 Parallel Programming Models

In this section we will give a brief overview on the current parallel programming models. Parallel programming is necassary for reaping the benefits of hardware improvements. There are many models for parallel programming, but they can be roughly divided into two classes.

2.2.1 Process Interaction

In the parallel programming terminology, some problems (e.g. embarrasingly parallel problems) do not require communication. These problems can be divided into self contained subtasks. In that case there is no need for process interaction. However, this case is rare in real world tasks where many modes of dependencies need to be communicated between processes. Problems like weather prediction, heat diffusion etc. cannot be calculated in parallel without process interaction. Communication can be done explicitly or implicitly.

Shared Memory

Shared Memory model is the most common way to tackle communication on todays multi-core cpus. It is fast and does not incur communication overhead with remote memory locations. Only considering the process communication without threads,

processes share a common memory address space that they read and write to asyncronously. In the case of concurrent access, syncronization needs to be implemented to avoid race conditions. Common interactions shape the execution such as waiting, yielding or notifying. Syncronization can be expensive if the blocking of other processes takes too long therefore limiting gains of parallel programming. Considering threads as well the situation looks similar with one difference: threads share the memory of a single process. This means the threads are not in race condition with other threads of another process. Threads can be user level or kernel level. Changing the execution to another thread is cheap.

One interesting area of parallel programming is the development of lock-free data structures. In this case concurrent access to the data is not blocking, eliminating the block and wait operations. Depending on the memory model more care needs to be taken in order to guarentee correctness of lock-free algorithms. In the strong memory model (sequential consistency model) limits the possible reordering of the memory related to the execution by enforcing memory fences. Memory fence enforces the order of memory executions. On a high level we can imagine 4 memory barriers:

- LoadLoad: prevents reordering of loads performed before the barrier with loads afther the barrier.
- LoadStore: processor is allowed to skip load operations if the coherence still applies afterwards.
- StoreLoad: the latest stored value is visible to all other processors, loads after the barrier receive the correct value.
- StoreStore: prevents the reordering of the stores performed before the barrier with stores after the barrier.

Each restricting the reordering of the memory operations. In the strong memory model, all but StoreLoad reordering are prohibited, even though it acts as like all of the barriers combined. This way the compiler and the cpu cannot reorder the execution with more freedom, therefor limiting the possibilities of more relaxed memory operations.

With the weak memory model it is possible to use all of the reorderings. This way the hardware is released of the burden of implicit acquire and release of the memory (locking the memory essentially). Another important concept is the atomic compare-and-swap (CAS) operation. In most modern hardwares CAS operation is implemented on the hardware atomically. Introduced in C++ 11 the compare exchange weak allowing weak CAS operation, thus eliminating the need for a lock. Being able to use lock-free memory operations with cohesion guarentees lay the foundation of lock-free algorithms. In the context of parallel programming, not waiting for data is the key.

In this case the main focus of the scheduler is to schedule threads in a nonblocking manner, therefor it is very important to make use of weak memory model as well. Shared memory ensures low latency to memory access, but for cache locality the scheduler needs to keep the NUMA principle in mind.

Message Passing

In the message passing model, processes have their own private memory, and communication is performed by sending and receiving messages. This model is well-suited for distributed systems, where processes may run on different machines or processors without shared memory. MPI (Message Passing Interface) is a common standard used in this model, which ensures scalability and fault tolerance in large-scale systems via explicit .

Message Passing solves the limitation of scalability in Shared Memory models, though while having more overhead in execution planing. Newest research is in asyncronous communication primitives primarily after MPI-3 using nonblocking collective operations in MPI-3, fault tolerance for exascale computing with User-Level Failure Mitigation (ULFM) proposal for MPI and automatic optimization of hybrid programming models (e.g. MPI + OMP).

Message Passing schedulers have to face the communication overhead of the model. Selecting the optimal nodes/clusters for communication is a challenge itself. Because scheduling latency is not a big consideration for exascale type of work, the focus is the saturation of computing power as well as data coherence (fault tolerance).

Partitioned Global Address Space (PGAS)

PGAS is a hybrid approach combining the shared memory aspect by logically partitioning the distributed memory. The whole memory is adressible to the process without the need for message passing. Although internal implementation uses memory management similar to message passing. This often involves moving memory to caches of the clusters and/or propagating the updates in the system. Popular examples are Chapel[**chapel**], X10 and Unified Parallel C. Current research includes more efficient runtime enviroments with cheaper data movements, interoperability with existing frameworks such as MPI, and better scaling for exascale computing.

2.2.2 Problem Decomposition

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as

decomposition or partitioning. There are two basic ways to partition computational work among parallel tasks: data parallelism and task parallelism.

Task Parallelism

Task parallelism is a parallel programming model that focuses on the concurrent execution of different tasks or processes. Unlike data parallelism, which applies the same operation to multiple data elements simultaneously, task parallelism emphasizes the distribution of distinct operations across multiple processing units. In task parallelism, a problem is decomposed into separate tasks that can be executed concurrently. These tasks often have different behaviors and may require inter-task communication, making it well-suited for problems with diverse subtasks. Task graphs or Directed Acyclic Graphs (DAGs) are commonly used to represent task dependencies and guide execution order. Key considerations in task parallelism include effective load balancing, managing task granularity, and efficient scheduling. Work stealing algorithms are often employed to dynamically balance workload across processors. Popular frameworks supporting task parallelism include OpenMP tasks, Intel Threading Building Blocks (TBB), Cilk Plus (Deprecated) and Taskflow.

Task parallelism, particularly when implemented with work stealing, offers significant advantages in load balancing and scalability. Work stealing dynamically redistributes tasks among threads, reducing idle time and improving overall system utilization. This approach is especially effective for irregular workloads or when task execution times are unpredictable. However, task parallelism faces several challenges. In work stealing, the act of stealing can introduce cache coherence issues, especially in NUMA architectures. When a task is stolen, its data may need to be transferred between NUMA nodes, incurring latency penalties. Moreover, frequent stealing can lead to cache thrashing, negatively impacting performance. Synchronization mechanisms like mutexes, while necessary for maintaining data consistency, can become bottlenecks. Threads may experience significant waiting times due to lock contention, particularly in systems with high core counts. This can lead to decreased parallelism and increased latency, again showing the importance of lock-free algorithms. Task granularity presents another challenge. Too fine-grained tasks increase scheduling overhead, while too coarse-grained tasks limit parallelism. Achieving the right balance is crucial for optimal performance. Lastly, the effectiveness of work stealing can be limited by task dependencies. Complex dependency graphs may restrict stealing opportunities, potentially leading to load imbalances and reduced parallel efficiency.

Data Parallelism

Data parallelism is a parallel programming model that focuses on distributing data across multiple processing units, which then perform the same operation on different data subsets simultaneously [**Hillis1986**]. This approach is particularly effective for problems with regular data structures, such as arrays or matrices, where the same computation can be applied independently to many data elements [**Blelloch1990**].

One of the key advantages of data parallelism is its scalability. As the data size increases, more processing units can be added to maintain or improve performance [**Darema2001**]. This model also often leads to simpler program structures, as the same code is replicated across processing units, reducing the complexity of parallel programming [**Chamberlain2007**].

However, data parallelism faces challenges in load balancing, especially when data is non-uniformly distributed or when computations on different data elements take varying amounts of time [**Yelick1998**]. Communication overhead can also become a bottleneck, particularly when data needs to be exchanged between processing units or when dealing with distributed memory systems [**Dinan2009**].

Modern implementations of data parallelism often utilize SIMD (Single Instruction, Multiple Data) instructions available in many processors, allowing for efficient vectorization of computations [**Fog2017**]. GPU computing has also embraced data parallelism, offering massive parallelism for suitable problems [**Owens2008**].

Recent research in data parallelism focuses on improving its applicability to irregular problems, enhancing load balancing techniques, and integrating it with other parallel programming models for hybrid approaches [**Kulkarni2007**].

Bibliography

[1] Hillis, W.D., Steele Jr, G.L.: Data parallel algorithms. Communications of the ACM 29(12), 1170-1183 (1986)

[2] Blelloch, G.E.: Vector models for data-parallel computing. MIT press (1990)

[3] Darema, F.: The spmd model: Past, present and future. In: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, pp. 1-1. Springer, Berlin, Heidelberg (2001)

[4] Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. The International Journal of High Performance Computing Applications 21(3), 291-312 (2007)

[5] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. Concurrency: Practice and Experience 10(11-13), 825-836 (1998)

[6] Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1-11 (2009)

[7] Fog, A.: The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering (2017)

[8] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96(5), 879-899 (2008)

[9] Kulkarni, M., Burtscher, M., Cascaval, C., Pingali, K.: Lonestar: A suite of parallel irregular programs. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 65-76. IEEE (2009)

Stream and Implicit Parallelization

2.1 Low Latency Scheduling

2.2 Cache Aware Scheduling

2.3 Concurrent Data Structures for Scheduling

2.4 Application Domains for Low Latency Scheduling

3 Developed Architecture

¡aaaaaaaaaaaa

Abbreviations

List of Figures

List of Tables