CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Performance-targeted Resource-aware Task Scheduling for Heterogeneous Platforms

Master's thesis in EMBEDDED ELECTRONIC SYSTEM DESIGN

Agnes Rohlin
Henrik Fahlgren

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Performance-targeted Resource-aware Task Scheduling for Heterogeneous Platforms

AGNES ROHLIN
HENRIK FAHLGREN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Performance-targeted Resource-aware Task Scheduling for Heterogeneous Platforms
AGNES ROHLIN
HENRIK FAHLGREN

Supervisor: Miquel Pericàs, Computer Science and Engineering
Examiner: Per Larsson-Edefors, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Performance-targeted Resource-aware Task Scheduling for Heterogeneous Platforms
AGNES ROHLIN
HENRIK FAHLGREN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Embedded environments often come with strict energy requirements, especially in mobile devices. Heterogeneous architectures are often used in these systems as they provide resources useful in different scenarios. However, introducing more resources requires proper scheduling to be able to utilize the resources efficiently. XiTAO is a resource-aware runtime which dynamically allocates appropriate resources to provide interference-free execution. This thesis aims to extend the XiTAO runtime to consider heterogeneous scheduling as it currently only considers scheduling for a homogeneous platform. For this thesis, only architectures with heterogeneous cores using the same instruction set architecture are considered. Specifically, a Huawei processor with ARM big-LITTLE architecture nodes mounted on a Hikey960 board is used. After surveying related work, four scheduling extensions are presented and evaluated in the thesis. Two extensions target the critical path of the application of which one schedules tasks on the critical path on predefined big cores while the other uses a history-based method of finding the most suitable cores. The third extension uses a history-based method to recognize which tasks will get the greatest performance increase from running on a big core compared to a LITTLE core. The fourth extension changes the amount of resources given to each task dynamically by observing the load and history of the system together with a moldability feature of XiTAO. The last extension can be applied simultaneously as the other scheduling extensions or individually. The scheduling extensions were evaluated with synthetic benchmarks with three different kernels, highlighting specific scenarios. The results show a speedup up to 29% for a randomized directed acyclic graph with our extensions compared the original runtime scheduling.

Keywords: runtime scheduling, heterogeneous hardware, computer science

Acknowledgements

We would like to thank our supervisor, Miquel Pericàs, for letting us take his runtime apart and for providing continuous technical support. We would also like to thank the members of the 96boards forum site for insight in any and all hardware problems.

Agnes Rohlin, Gothenburg, May 2018
Henrik Fahlgren, Gothenburg, May 2018

Acronyms

API - Application Programming Interface

AQ - Assembly Queue

DAG - Directed Acyclic Graphs

DPA - Dynamic Place Allocation

DVFS - Dynamic Voltage Frequency Scaling

HCMP - Heterogeneous Chip Multi Processor

ISA - Instruction Set Architecture

STA - Software Topology Address

TAO - Task Assembly Object

UART - Universal Asynchronous Receiver/Transmitter

UEFI - Unified Extensible Firmware Interface

WSQ - Work Stealing Queue

Contents

1 Introduction **1**
1.1 Project Aim 2
1.2 Scope 2
1.3 Thesis Outline 3

2 Technical Background **5**
2.1 Scheduling 5
2.1.1 Graph Theory 5
2.1.2 Scheduling Goals and Important Properties 6
2.1.3 Scheduling Techniques 7
2.2 XiTAO 8
2.2.1 Task Assembly Objects 9
2.2.2 Elastic Places and Dynamic Place Allocation 10
2.2.3 Low-Level Programming API 13
2.3 Scheduling in a Heterogeneous Environment 14

3 Scheduling Implementation **17**
3.1 Performance Trace Table (PTT) 17
3.2 Heterogeneous Scheduling Extensions 18
3.2.1 Criticality-based Scheduling 19
3.2.2 Weight-based Scheduling 21
3.3 Task Molding 21

4 Evaluation Method **23**
4.1 Evaluation Platform 23
4.1.1 Linux Installation 23
4.2 Evaluation Benchmarks 24
4.2.1 Single and Parallel Chains 25
4.2.2 Fork-Join 25
4.2.3 Randomized DAG 26
4.2.4 Kernels 27
4.3 Evaluation Metrics and Methodology 31

5 Results **33**
5.1 Parallel Chains Evaluation 34
5.2 Fork-Join Evaluation 35
5.3 Randomized DAG Evaluation 36

5.4 Task Molding Evaluation 40

6 Discussion **41**
6.1 Results 41
6.2 Evaluation Environment 42
6.3 Design Choices 43

7 Conclusion **47**

Bibliography **49**

A Linux Installation Guide **I**

1 Introduction

The fundamental limitations of device scaling and the energy concerns that follow are, as already known, a first-order concern today. Striving for energy-efficiency is unsurprisingly a high priority interest. Today, there are complex modern computing platforms packed with substantial amounts of resources which could be efficiently exploited to achieve performance gains and energy savings. Here, parallelism is not enough to utilize the packaged resources efficiently. Tuning the usage of resources such as caches, memory bandwidths, processor pipelines and power to properly exploit the platform and to avoid destructive interference across tasks is critical, especially for heterogeneous architectures in embedded environments.

Heterogeneous architectures combine several *non-identical* processors or cores, often also referred to as asymmetric architectures/cores. The advantage of exploring heterogeneity is that utilizing different characteristics and properties can improve various aspects of computing significantly. However, the cost of utilizing these heterogeneities, similarly to the introduction of multicore, is that of complexity in management and scheduling.

Embedded environments often come with strict energy requirements, for example in battery operated devices where heterogeneous architectures can be of great interest. Heterogeneous architectures and corresponding task scheduling within mobile devices is already a researched area where processors such as ARMs big.LITTLE with its global task scheduling are proven successful [1]. The fundamental idea for the software support of the big.LITTLE is that executing code should be allocated to and run on an appropriately sized core to provide the performance needed for the task dynamically (using migration), effectively achieving great energy savings suitable for mobile environments [1].

In general, scheduling for heterogeneous processors has been thoroughly researched and interesting schemas utilizing different approaches and techniques have been developed. For example, Heterogeneous Earliest-Finish-Time (HEFT) scheduling schema by Topcuoglu et al. [2], is an offline static scheduler that ranks tasks based on computation and communication costs before scheduling them on the most suitable processor resulting in the earliest finish for each task. Another example is the dynamic criticality-aware scheduling by Chronaki et al. [3], which dynamically assigns tasks to cores based on runtime information without the need for profiling.

Scheduling of parallel architectures has in the past aimed to achieve a high CPU utilization (greedy scheduling), always keeping all resources busy for the biggest performance gain. This approach worked well when memory bandwidth was ample and caches private, but in the shared cache architectures of today together with the trending memory wall, it results in unwanted interference causing performance losses.

This project deals with integrating heterogeneous scheduling into the task-based runtime system called XiTAO. XiTAO provides a solution to the greedy scheduling problem with a goal of interference-free scheduling [4], [5]. The XiTAO runtime generalizes the concept of tasks to achieve a better mapping and dynamically allocates hardware resources to them, which allows for interference-free scheduling with high granularity of parallelism without the scheduling overhead. The runtime already utilizes resource-aware scheduling by providing resource hints to the runtime and allocates the appropriate amount of resources dynamically allowing for both resource efficiency and avoidance of destructive interference. However, the resource aspect considered is that of a homogeneous architecture, mainly the number of cores suitable to use.

This project is being conducted in the scope of the EU project LEGaTO(Low Energy Toolset for Heterogeneous Computing) with Project ID 780681, where the aim is to provide a tool set for efficiently managing heterogeneous computing [6]. Chalmers University of Technology will explore high-risk and high-reward possible energy savings with the XiTAO runtime which will be integrated into the common runtime for the LEGaTO project if proven successful.

1.1 Project Aim

As previously stated, this project aims to integrate further resource-aware scheduling into the task-based runtime system, XiTAO, to properly exploit heterogeneous hardware for an increase in performance. The hardware considered in this project is heterogeneous chip multiprocessors (HCMP) where all cores utilize the same instruction set architecture (ISA) but provide different capabilities.

1.2 Scope

The project consists of identifying the resources available on the hardware, adjusting the scheduler to benefit from a heterogeneous platform and properly evaluate the performance gains.

First, the available resources must be identified to introduce some visibility of the platform to the runtime. The resource information should be accessible for the runtime, tangible, quantified and preferably be gathered automatically. The runtime should have some knowledge of the hardware it is currently operating on in order to make efficient scheduling decisions.

Secondly, the XiTAO runtime has to be extended to consider the added resources. To efficiently exploit heterogeneous hardware, adjusting the scheduling decisions to that of heterogeneous scheduling is necessary. For this, different schemas and methods have to be considered to find a suitable solution based on applicability to the runtime and effectiveness. In this project, we focus on heterogeneity where cores utilize different capabilities providing the same functionality with the same ISA, such as the big.LITTLE architecture. Specifically, we limit ourselves to using the HiKey 960 board [1] during this project.

A final key part of the problem is to properly evaluate the extended scheduler. The scheduler has to be evaluated in terms of performance and compared to the performance of the non-extended runtime. The metrics considered are limited and the performance of the scheduling is measured mainly in throughput. We also limit the evaluation stage to synthetic benchmarks only. The synthetic benchmarks should test bottleneck scenarios with high load such as intensive cache usage or heavy computations.

1.3 Thesis Outline

Following this introduction, Chapter 2 covers some essential scheduling theory and explains the fundamental concepts of the XiTAO runtime before surveying a couple of relevant heterogeneous scheduling approaches which we base our implementation on. The scheduling implementation is then presented in Chapter 3. In Chapter 4, the evaluation methodology is presented before the results are presented in Chapter 5. Finally, we discuss the results, the evaluation process and our design choices in Chapter 6 before reaching a conclusion in Chapter 7.

[1]https://www.96boards.org/product/hikey960/

2 Technical Background

This chapter provides the technical background necessary for understanding the work carried out in this thesis. It also serves as background for decisions taken for our scheduling implementation. Basic graph theory and scheduling methodology is first explained in Section 2.1 before the runtime is introduced in Section 2.2. Finally, some current heterogeneous scheduling techniques are presented in Section 2.3. The final section forms the basis for the policy in our scheduling implementation.

2.1 Scheduling

The following section covers basic scheduling definitions and theory which are relevant to understand this thesis. First, some basic graph theory is introduced in Section 2.1.1, then some common goals and interesting properties of scheduling are demonstrated in Section 2.1.2 and lastly a couple of basic scheduling techniques is explained in Section 2.1.3.

2.1.1 Graph Theory

Representation of scheduled execution flows are often represented with graphs. The basic notation relevant to recall is that a *graph* G is a pair of *nodes* and *edges*, $G = (V, E)$, where edges are denoted by the two nodes it connects $(i, j) \in E$ [7]. A *directed* graph is a graph where the edges have an associated direction, $G = (V, A)$ where A denotes a set of *arches*, $(i, j) \in A$ where i precedes j. Two example graphs with and without direction are shown in Figure 2.1. Nodes without predecessors are called entry nodes and nodes without a successor are called exit nodes. A node $i \in V$ is associated with a degree, $deg(i)$, which is the number of edges associated to it. To simplify we can distinguish between a nodes input edges, $indeg(i)$ and output edges, $outdeg(i)$.

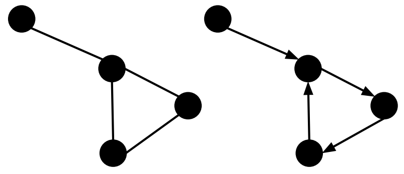


Figure 2.1: An illustration of the difference between an undirected (left) and a directed graph (right).

Specifically, the graph class *directed acyclic graph* is common in scheduling and a central part of the representation of applications in the XiTAO runtime. A directed acyclic graph (DAG) only has nodes that are not reachable from itself, it has no *cycles* [7]. This pattern of precedence of nodes is a convenient representation of precedence of tasks in a load to be scheduled. Finally, the *critical path* is the longest path in the graph (without cycles) where the definition of longest and distance can differ. A DAG is illustrated in Figure 2.2 where the critical path is marked with dotted lines.

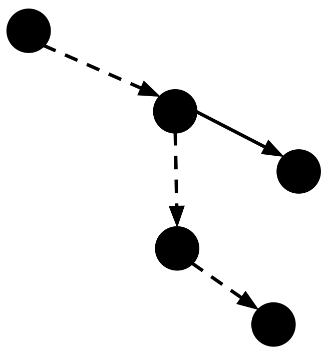


Figure 2.2: An illustration of a DAG with a critical path in dotted lines.

2.1.2 Scheduling Goals and Important Properties

To evaluate the performance of a scheduler it is necessary to have an idea of what the scheduling should achieve. The goals of the scheduler heavily depend on the environment, although some goals are more common than others [8]. Policy enforcement, fairness and balance are generally important goals to consider and depending on the environment, the context of these varies. Additional goals stated by [4], [8] can be for example:

- **Throughput** - Maximize work done per time unit.
- **Turnaround time** - Minimize time between start and finish.
- **CPU utilization** - Keep the CPU busy with work (minimize idle time).
- **Scalability** - Scale with increasing number/types of resources.
- **Low overheads** - Minimize the scheduling overhead.
- **Communication bandwidth** - Minimize total communication bandwidth.
- **Priority based** - Prioritize specific tasks, i.e. tasks on the critical path.
- **Bind the memory and stack** - Ensure that neither memory or stack overflow.
- **Minimize work-time inflation** - Minimize the additional time spent beyond the equivalent required work time in a sequential computation.
- **Migration cost** - Minimize migration cost between different resources.

Naturally, all of these goals may not always be of interest. Some of the goals strongly correlate while others might be conflicting. An example of this has already been highlighted where the goal of CPU utilization can conflict with throughput due to interference in a multicore system. In addition to the goals of scheduling, other properties also affect the implementation of the scheduler. Some additional properties related to scheduling relevant to this thesis are parallel granularity, destructive interference/cooperative sharing and locality.

Parallel granularity is the amount of work carried out by a thread between two scheduling points [5]. For a fixed problem size, the granularity decreases (becomes more fine grained) when increasing the number of cores if the work is split evenly among the cores [9]. In scheduling, a frequently seen trade-off is that of coarse and fine-grained tasks. By coarsening the parallelism of tasks, the overhead associated with scheduling decreases while a finer grain of task parallelism increases the possible execution parallelism resulting in a higher CPU utilization.

Another already mentioned important property that affects the performance in a shared cache processor is **destructive interference and cooperative sharing** [9]. In cooperative sharing, memory blocks brought into a shared cache by one core are also used by the other cores, reducing the number of memory accesses. The opposite case is destructive sharing/interference where cores need to access different blocks and are continuously replacing the blocks in the shared caches, resulting in more memory accesses. These two concepts are important to consider in the case of shared cache environment, as it can greatly affect performance outcome.

Locality is a property of memory accesses in program executions [9]. Temporal locality is the property of accessing the same memory addresses repeatedly while spatial locality is the property of accessing memory addresses which are close in address space. In an environment where main memory and caches are structured into blocks, spatial locality is converted into temporal locality as a block or a page is fetched. Exploiting these proprieties of a process can heavily impact the memory performance. Locality can be exploited by adhering to affinity by running related processes on the same resources and/or resources sharing other resources.

2.1.3 Scheduling Techniques

Scheduling techniques and algorithms are relevant in various environments ranging from scheduling in operating systems and embedded runtimes to network schedulers etc. Depending on the environment and the goals of the scheduling, there are many aspects to consider such as dynamic or static scheduling, automatic or manual scheduling etc. For our thesis we are interested in the case of multicore scheduling where the scheduling should be done dynamically by the runtime. The difficulty of multi-threaded scheduling is that there are essentially two dimensions to consider, what thread/process to run and on what resource [8]. This, in comparison to a single-core environment, complicates the scheduling as it allows for more scheduling goals to be targeted and by extension more trade-offs. The following section presents some fundamental concepts and relevant techniques for scheduling in general.

Essentially, one can consider two basic workload sharing perspectives in a multicore environment, **time sharing** and **space sharing** [8]. Time sharing is scheduling where a free core receives first available job from a shared data structure of jobs. This results in a balanced load but may result in contention for the data structure and context switch overheads. In addition, it is also limited in scalability and does not naturally adhere to processor affinity. Space sharing, is instead, scheduling a process at the same time across multiple cores. A task is scheduled first when all related jobs can be scheduled. This way affinity and locality can be targeted, and the scheduling overhead is smaller, but it introduces idle time into the system. A hybrid technique is gang scheduling where groups of related threads are scheduled as a gang and all members of a gang run on a shared time slot on different resources [10]. Each resource is scheduled synchronously with discrete time quotas and at the start of each quota every resource is rescheduled. If a thread blocks, the resource stays idle until the end of the time quantum.

The above perspectives all share the idea of a global scheduler managing the load of a system. Having a centralized scheduler can limit the scalability of a system as the number of resources grow. Additionally, if scheduled processes or threads spawn new threads, the result could be an unbalanced workload for the system. Two often adopted load balancing dynamical schedulers are work sharing and work stealing.

In **work sharing**, newly generated tasks are balanced by the scheduler as it migrates some of the tasks to other resources, hopefully underutilized cores [11]. This can be implemented using a centralized task pool, which is a simple but not scalable implementation, or with distributed local task pools where tasks can balance in parallel whenever a resource is idle.

In **work stealing**, underutilized resources attempt to steal tasks from other resources when they run out of tasks to execute [12]. This can be implemented as a distributed implementation which scales well. In addition, it results in fewer task migrations than work sharing, due to only stealing when necessary.

2.2 XiTAO

XiTAO is an execution model and runtime developed for resource-efficient interference-free execution in a multicore environment and is built on top of C++11 [4], [5], [13]. The basic idea of XiTAO is to minimize the overheads associated with fine-grained parallelism without sacrificing the performance gains by encapsulating parallel set of tasks into generalized tasks. In addition, the generalization of tasks allows for interference-free scheduling by avoiding oversubscribing the resources. XiTAO does this by generalizing tasks into a set of tasks with an internal scheduler and a resource hint. These generalized tasks, called *Task Assembly Objects* (TAOs), are moldable entities allocated to a suitable hardware place directed by its resource hint. The concept of TAOs is further explained in Section 2.2.1. A resource hint within a TAO allows for the runtime to treat TAOs as moldable and schedule them onto *elastic places*. Elastic places are hardware places which can be dynamically provided to TAOs at runtime. Additionally, this allows for locality-aware and interference-free scheduling as tasks can be bound within a TAO and scheduled onto adjacent cores using elastic places. Elastic places and the Dynamic Place Allocation (DPA) scheduler, which is responsible for allocating resources, as well as the governing distributed scheduler, are further explained in Section 2.2.2. Locality between TAOs can also be targeted in the runtime as DAGs of TAOs can be mapped with a virtual software topology schema. This is done by assigning a software topology address (STA) to a TAO from an n-dimensional space (topology) selected by the application. The runtime translates the STA into a distinct hardware place This way two

TAOs that share an STA correlates to having a high degree of locality and should be scheduled accordingly. The concept of software topology is not further employed by any scheduling in this thesis and will not be further explained. Finally, to use the runtime, a low-level programming API is enforced where the tasks are encapsulated into TAOs and a resource hint is provided, this is explained in Section 2.2.3.

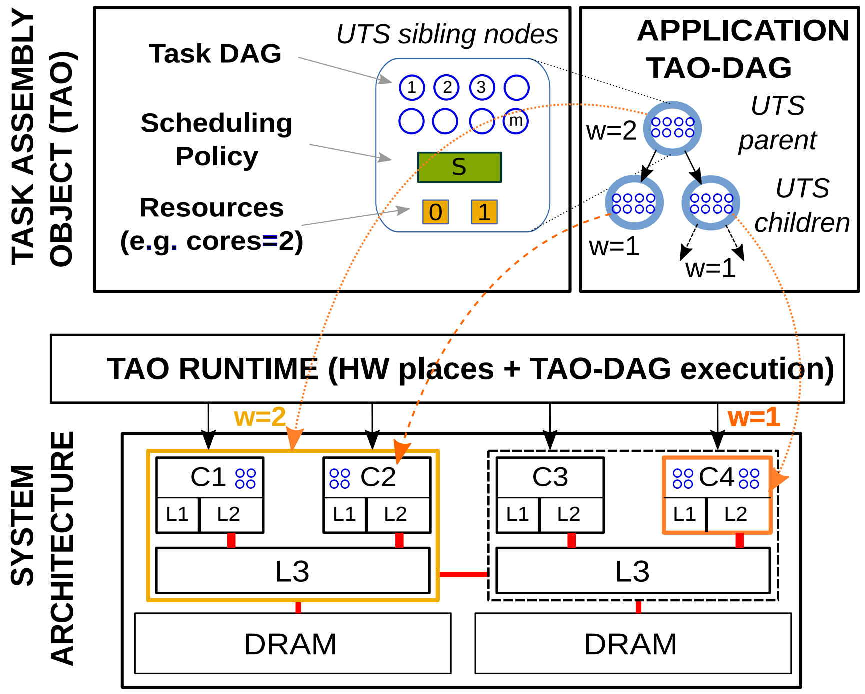


Figure 2.3: A concept image of the XiTAO runtime architecture running a benchmark UTS. Reproduced with permission from [13].

2.2.1 Task Assembly Objects

The two-dimensional compute granularity is achieved by aggregating tasks and worker threads onto cores in Task Assembly Objects (TAO) and effectively scheduling these [4]. As mentioned, a TAO is a moldable entity consisting of fine-grained tasks, an internal scheduler and a resource hint. A representation of a TAO can be seen in Figure 2.4. The generalization allows for a TAO to hint at a resource requirement while providing its own internal scheduler to execute either a simple task or nested tasks onto these cores. The fine-grained tasks are the work to be done, either as multiple tasks (with or without dependencies) or as a simple task, the internal scheduler executes the TAO on a set of *virtual* cores as suggested by the resource hint. The resource hint can be in reference to resources such as cores or caches but is at runtime translated into how many cores is desired. The governing scheduler simply sees the TAOs as ”wider” units of computation, a black box, filled with work ranging from simple tasks, dependent set of tasks (DAGs) to possibly even whole runtimes (OpenMP, Intels TBB). By doing this, the runtime spends less time on global scheduling and destructive interference is avoided.

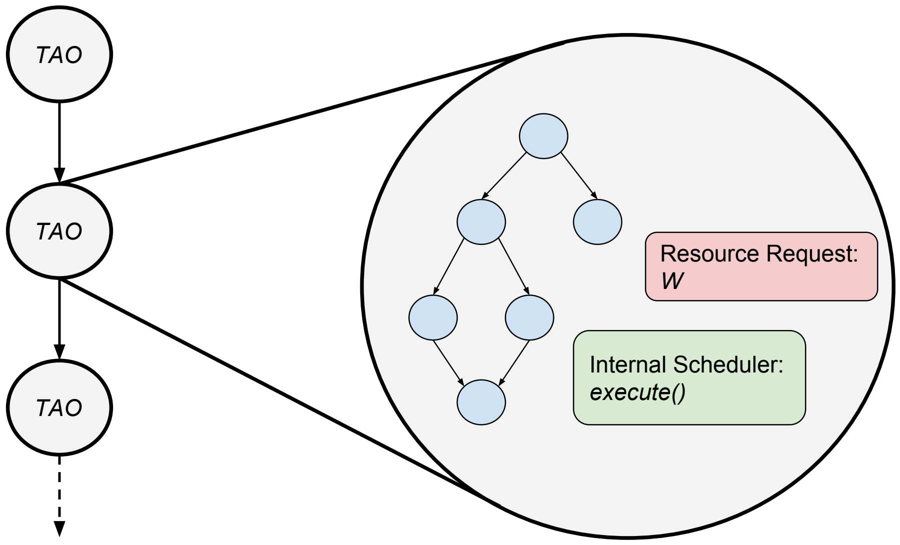


Figure 2.4: An instance of a task assembly object (TAO) contains tasks or some work to be done, an internal scheduler invoked by a function call, `execute()`, and a requested resource hint to how many resources should be used for this TAO.

2.2.2 Elastic Places and Dynamic Place Allocation

To effectively achieve scalability and performance portability, the XiTAO runtime must support the moldability of TAOs, by dynamically assigning appropriate hardware partitions that provide necessary resources [5]. Due to system changes and runtime conditions appropriate hardware partitions can change and the runtime must be able to aggregate and disaggregate resources dynamically, this is the basis for elastic places. Elastic places are resource containers, dynamically constructed partitions of available resources where TAOs can execute, which give controlled resource sharing and protection against interference. The idea is to allocate a new hardware place (resource container) with the required (based on the resource hint) resources when a new TAO is scheduled. This hardware place becomes the execution place for the TAO as it is time to execute. The resource hint is effectively translated into a corresponding resource width.

In the XiTAO runtime, the DPA and load balancing are implemented in a decentralized worker-loop using distributed queues and work stealing [5]. The runtime implements two sets of queues for ready objects, work-stealing queues (WSQ) and assembly queues (AQ), the queues and the dynamic between them is show in Figure 2.5. A work-stealing queue is a ready queue where workers can steal objects from each other. The runtime follows the Cilk5 work-stealing policy on top of the work-stealing queues. The Cilk5 work-stealing policy is a global work-stealing policy of FIFO order where newly created tasks are pushed and retrieved in a LIFO manner [14]. When a TAO is retrieved from the work-stealing queue it is distributed using the DPA into assembly-queues. The DPA is implemented by placing pointers to decoded TAOs, taken from the work-stealing queues into a set of assembly queues for the workers belonging to the hardware place. The worker distributing the TAO will always select a compact and sequential set of workers along with itself according to the resource hint. The TAO will then wait in the assembly queues until its asynchronously fetched for execution. Deadlock is avoided by forcing FIFO ordering on the assembly queues through atomic distribution-operations of the TAOs.

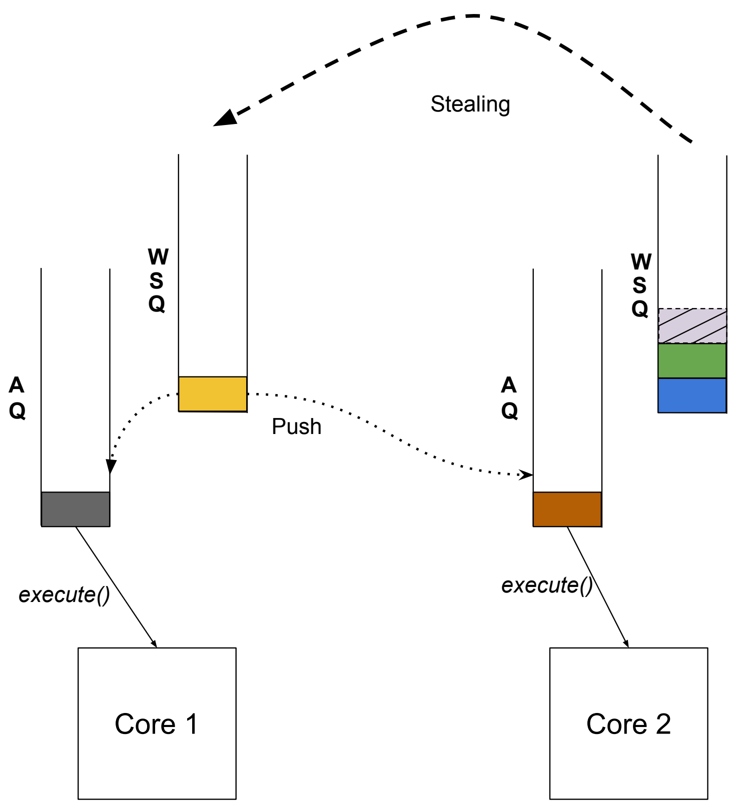


Figure 2.5: An illustration of the distributed scheduling queues used in XiTAO. Each thread mapped to a core maintains a WSQ and an AQ of ready TAOs. TAOs can either be fetched from the AQ to be executed or fetched from the WSQ to be distributed among threads WSQs. Stealing is allowed for load balancing between the WSQs.

As the runtime is initialized, a desired number of threads are created and pinned to cores. These worker threads handle all scheduling and execution by fetching work in a worker loop, effectively formulating the governing distributed scheduler. Pseudo code for the `while(1)` worker loop can be seen in Listing 1 and consists of six steps. First, it checks for a forwarded TAO from the previous iteration which should be scheduled with the DPA. Second, it checks for work in the assembly queue and third, if work was found, it executes it. The fourth step checks the work-stealing queue for work to be distributed and fifth, if no work was found it tries to steal. Finally, in the sixth step, it checks if there is any work left at all, if not, it terminates.

Listing 1: Worker loop making up the governing distributed scheduler.

```
int worker_loop(nthread){
    while(1){
        //1. Check forwarded path.
        //If there is a forwarded path then the previous iteration
        //of the worker loop resulted in a new ready TAO which needs
        //to be assigned an elastic place by the DPA scheduler.
        if(pt){
            assign(pt,nthread);
            pt=nullptr;
        }
        //2.Check own AQ. Look for a TAO in the AQ to be executed.
        if(!pt){
            pt=AQ.pop_front(nthread);
        }
        //3.If a TAO was found execute it by starting the execution
        //of its inner scheduler with the TAOs execute()
        //method. If the thread is the last worker for this TAO
        //(the TAO is completed), it commits the TAO and wakes up
        //a depending TAO.
        if(pt){
            pt->execute(nthread);
            //If last in TAO, commit and wakeup
            if(pt->last){
                new=pt->commit_and_wakeup();
                pt->cleanup;
            }
        }
        //4. Check local WSQ for work.
        if(!pt){
            pt=WSQ.pop_front(nthread);
        }
        //5. Try to steal some work from another WSQ if no work
        //was found.
        if{!pt}{
            pt=steal_work();
        }
        //6. If no work was found in any WSQ, check if there are
        //no more pending TAOs at all to run, if so, then terminate
        if(pending_tasks==0){
            break;
        }
    }
}
```

2.2.3 Low-Level Programming API

As XiTAO is intended to be used as a backend for models such as OpenMP or Intels TBB, where the computations are portrayed in DAGs, the current low-level programming API allows for explicit declaration of TAOs and TAO-DAGs [5]. Thus, an application in XiTAO is currently expressed as a dynamic TAO-DAG with edges to indicate ordering-dependencies between TAOs. A TAO-DAG with single threaded TAOs is conceptually equal to a classic task DAG. TAOs can be designed either as a set of tasks with dependencies or as a simple task through a class creation where XiTAO provides low-level constructs to represent TAO objects as C++ classes. A minimal TAO class is a derived AssemblyTask class and is initialized with a resource hint and possibly an address in the software topology. It must also contain a `cleanup()` and an `execute()` function which are called by the worker threads as they execute and commit TAOs. The execute function either contains a simple set of work which is readily executed by the thread, or an internal scheduler and a set of tasks to be executed. A minimal empty TAO can be seen in Listing 2.

Listing 2: An empty TAO class containing an initialization with a resource hint, a cleanup function and an execute function.

```
class TAO_name : public AssemblyTask
{
    public:
        //Initialization with a resource hint, either static
        //or dynamic. An STA address can also be provided
        //if not it will be resource 0.
        TAO_name() : AssemblyTask(resource_hint)
        {

        }

        int cleanup()
        {
        //Cleanup function after execution
        }

        int execute(int threadid)
        {
        //Internal scheduler or simple execution
        }
}
```

2.3 Scheduling in a Heterogeneous Environment

Task scheduling on a heterogeneous platform, contrary to a homogeneous platform, includes the problem of assigning the appropriate tasks to the most suitable cores. Many multicore scheduling approaches today assume equal performance. For example, dynamic scheduling techniques such as work-stealing or work-sharing does not consider the workload of tasks when scheduling which can result in unbalanced loads in a heterogeneous environment. Finding the optimal scheduling for an asymmetric multicore architecture is a NP-complete problem [15].

As a basis for our scheduling implementation, we have surveyed and gathered some relevant techniques. These scheduling methods are used as background information and inspiration for parts of our thesis.

Heterogeneous Earliest-Finish-Time (HEFT)

HEFT is a proposed novel method for heterogeneous task scheduling by Topcuoglu et al. [2]. The HEFT algorithm consists of ranking the tasks of a DAG in order of longest path to finish and then assigning the highest-ranking tasks to the core that will minimize the overall finish time. An analysis of the DAG is done to calculate the workload and communication cost of each node and edge before the tasks can be ranked. The tasks are then placed in a queue where the scheduler picks the top task and calculates which core will be able to finish this task earliest using insertion-based scheduling. HEFT works for various types of heterogeneous systems and assumes prior knowledge of the number of tasks and their properties.

Critical Path on Processor (CPOP)

Alongside HEFT, Topcuoglu et al. also proposed CPOP [2]. CPOP has a similar algorithm but with a different approach to ranking tasks. HEFT uses a downward ranking which starts at the exit node of the DAG, and for each node includes the maximum length of its successors to its own length. CPOP considers both the upward and downward rank when calculating the final ranks of tasks where the upward rank is calculation starts at the entry node and considers its predecessor instead. Both CPOP and HEFT are well established scheduling options for heterogeneous platforms, but HEFT is shown to perform better in most situations [16].

Heterogeneous Multicore Clustering and Duplicate Based on Down Succ_sum List Algorithm (HCDDSL)

HCDDSL is a method proposed by Cheng et al. [16], that uses a combination of scheduling methods. The first step of the algorithm is to cluster-analyze the DAG and create clusters of the tasks. The second step is to sort the tasks by their downward rank similarly to HEFT. Lastly, it allocates tasks to the cores to minimize finish time, once again like HEFT, apart from duplication. If the finish time can be minimized by copying the predecessor task, the task will be duplicated.

Criticality Aware Task Scheduling (CATS)

CATS is a dynamic scheduling approach where no prior knowledge about the workload of the tasks is assumed [3], [17]. Rather than using the load of the tasks as input when ranking the tasks, CATS solely uses the number of successors to find the critical path. The critical path is then put in a critical queue. Tasks from the critical queue are scheduled on high-performance cores and tasks from the non-critical queue are scheduled on lower-performance cores. This was evaluated on a board featuring an eight-core Samsung Exynos 5422 chip with ARM big.LITTLE architecture. To evaluate CATS, dynamic Heterogeneous Earliest Finish Time (dHEFT) is introduced as a reference. dHEFT uses the same principles as HEFT but instead of knowing the load of tasks prior to scheduling, discovers them at runtime. In the evaluation, a general performance improvement over the dynamic HEFT was found.

Bias Scheduling in Heterogeneous Multicore Architectures

A proposed method which focuses on single-ISA heterogeneous multicore processors and how different kinds of tasks perform on each core, is Bias Scheduling [18]. The main idea is to categorize applications into two groups: Applications gaining large speedup by running on a big core compared to a LITTLE and applications gaining modest speedup by running on a big core. The speedup is approximated by accessing hardware counters for stall cycles. Applications are then scheduled on big cores if they provide large speedup and on LITTLE cores if the speedup would be modest.

Performance impact estimation (PIE)

PIE is a proposed method by Van Craeynest et al. [19], which focuses on single-ISA heterogeneous multicore processors and how different type of tasks perform on each core. The characteristics of tasks are matched to appropriate cores by predicting the instruction level parallelism and memory level parallelism of the tasks. The tasks are then scheduled accordingly.

3
Scheduling Implementation

This chapter explains the changes we made to the scheduling in XiTAO as well as the design choices for the implementations. First, the performance trace table, hereby also referred to as PTT, implementation is explained Section 3.1. The performance trace table allows for resource visibility which can effectively be used to make more intelligent scheduling decisions. The implemented extensions that make a scheduling decision between resource types are then described in Section 3.2. Finally, a scheduling extension to exploit the notion of elastic places and moldable tasks is explained in Section 3.3.

3.1 Performance Trace Table (PTT)

To be able to dynamically affect the scheduling decisions based on the available resources, we implemented a *performance* tracing of TAOs at runtime and a table to record results. Although several scheduling implementations surveyed in Section 2.3 assume prior knowledge of task loads [2], [16], this is not applicable in our case where the runtime knows nothing of the task loads initially. A table was implemented for each TAO type where each core and resource-width pair records execution time. Since XiTAO is based on TAOs, it allows for a natural tracing of past executions as traces can be gathered for each TAO class rather than each task, effectively minimizing recording overheads.

The table was implemented as a static variable with corresponding get/set functions. The get and set functions were declared similarly to the `execute()` and `cleanup` using virtual function specifiers, therefore requiring a TAO class (an inheriting class) to explicitly define the function.

The table is, for each TAO, organized by $(corenumber) \times (resourcewidth)$ as seen in Figure 3.1. Due to the distributed implementation of the scheduler, the table is organized to optimally fit into cache lines where each core only accesses one cache line indexed with core number. For each entry, the execution time in seconds (doubles) is saved, recorded after the system clock with the c++ chrono time library. Updating the table is done with little overhead, approximately taking less than 1% of the total execution time (average execution time is 0.002s for our TAOs). The table is updated after each TAO execution with a weighted time of 1:4 to the old value of the table: $savedvalue = \frac{(4*oldvalue)+newvalue}{5}$.

The table is updated by the leader of the TAO to minimize cache overheads. In XiTAO the leader of a TAO is decided on when the TAO is distributed to the assembly queues by the DPA. The leader is then set to $(core/width) * width$, where *core* is the core distributing the TAO and width is the resource hint. The division is a floor division, making only some cores eligible to become leaders for large resource widths. For example, if core number seven were to distribute a TAO with resource width four, core 4 would be chosen as leader. For the PTT this means that every core can have a recorded value of the TAO with $width = 1$ but only every fourth core can have a recorded value of the TAO with $width = 4$. By only allowing the leader to save its execution time, the amount of accesses to the table is minimized, but it can potentially skew the recorded result as the leader might not have the most representative view of the execution time of that TAO i.e. it can have the least or the most amount of work. This is however dealt with as we are weighting the recorded values 1:4 and any particularly diverging values does not affect the table significantly. Although this results in an additional read of the table we found it more important to be resilient to divergent measurements as this table aids the scheduling decisions.

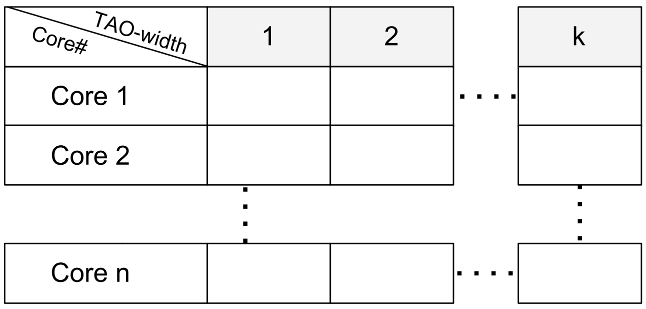


Figure 3.1: The structure of the performance trace table (PTT) where n is the number of cores in our system and k is the maximum resource width, $log2(\#cores)$.

This implementation of tracing execution history requires no knowledge of the available resources as the cores simply updates the corresponding index, independent of its resource type. This is beneficial not only for portability and potentially functional-heterogeneity, but it also allows for addressing temporally added heterogeneity such as dynamic voltage frequency scaling (DVFS) caused by heat variations.

3.2 Heterogeneous Scheduling Extensions

From the related work in Section 2.3, it is apparent that various approaches to heterogeneous scheduling has been researched. The presented schedulers explore different concepts in different environments with more or less success. We found that there were two directions for the scheduling that were appropriate for the heterogeneous environments of this thesis. There are those which target the critical path and ensure that the most critical work is run on a high-performance core and those which allocate tasks to different cores depending on performance differences of the tasks on the various resources. These two scheduling policies were both applicable for our environment and provided two different perspectives and focus points which could bring interesting results. Based on this, we present two different scheduler extensions to XiTAO, criticality-based and weight-based scheduling. The first one aims to target the criticality of a DAG and the second aims to exploit the performance difference between the TAOs on the two core clusters. The algorithms are inspired by the methods introduced in Section 2.3, especially Bias scheduling by Koufaty et al [18] and CATS by Chronaki et al. [17]. Contrary to the methods in Section 2.3, XiTAO is a distributed runtime without a central governing scheduler, thus we cannot implement identical solutions as CATS and Bias scheduling. Instead we focused on exploiting the benefits of XiTAO as we made use of the PTT to find suitable resources. The criticality-based scheduling and the weight-based scheduling extensions will be explained in detail in Section 3.2.1 and Section 3.2.2 respectively.

The scheduling extensions were implemented within the mechanisms of the runtime, specifically in the commit-and-wakeup mechanism. The commit-and-wakeup is a routine, within each TAO (or simple task), responsible for waking up depending tasks and is executed by the last core completing the execution of a TAO. Each call to the commit-and-wakeup checks which of the depending TAOs are ready for execution (all previous dependencies being committed). The ready TAOs are then placed into the work-stealing queues or forwarded locally. Making a scheduling decision within this mechanism allowed us to maintain the DPA and the notion of elastic places and resource allocation as it is, effectively preserving important key notions of the XiTAO runtime. None of the scheduling extension made any changes to the underlying load-balancing policy or to the DPA. While modifications of the work-stealing policy were considered, they proved less effective than the current simpler random work-stealing policy. The scheduling extensions were also implemented without utilizing the STA notion, briefly explained in Section 2.2, since the STA is a tool for targeting locality between TAOs and if used, the scheduler should allow for the TAOs to be scheduled according to the STA preference.

3.2.1 Criticality-based Scheduling

The CATS scheduling schema by Chronaki et al [17] was implemented as a pre-execution criticality-analysis where the critical path of the DAG to be executed is determined by the longest path and tasks are placed into two sets of centralized queues, critical and non-critical, depending on the criticality analysis. The decision of making the longest path the critical path proved effective and is suitable to our runtime environment where no knowledge of task load is available prior to execution. Therefore, our criticality scheduling extension in XiTAO also shares this notion of critical path.

For our implementation, the criticality analysis of the DAG is done as the runtime is started by calling a recursive function which assigns criticality values on the pushed TAOs and their successors. The recursive function traverses top-down through the DAG until it reaches the end node(s) and assigns each node a criticality value of $max(crit(child))$, effectively resulting in the first node of the longest path having the highest criticality value. An example of the criticality assignment of a small DAG can be seen in Figure 3.2 where the DAG has been traversed top to bottom recursively. Once each TAO is assigned a criticality value the runtime can start executing.

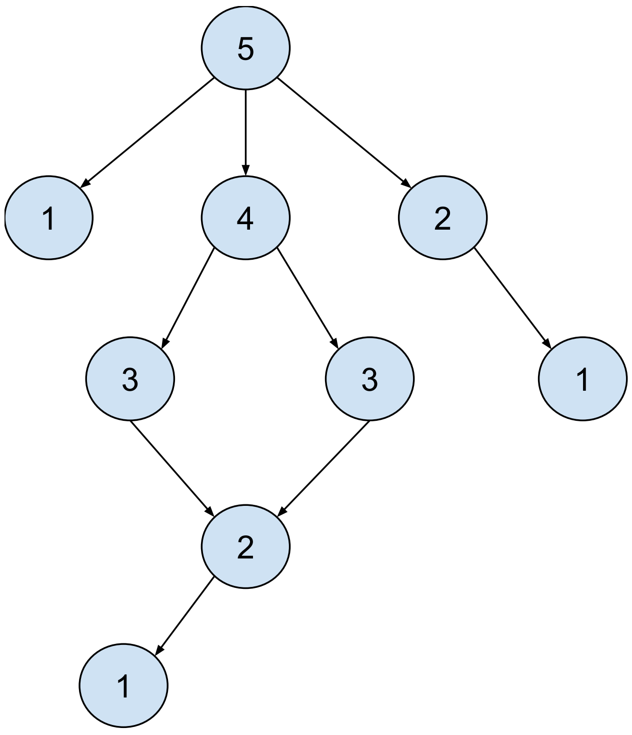


Figure 3.2: An example of the criticality values assigned to a DAG for the criticality-based scheduling extensions.

The criticality values are then used whenever a TAO is woken up in the commit-and-wakeup mechanism. When a TAO is being woken up, its criticality value is compared to the maximum criticality TAO that is currently running and the TAO is then deemed either critical or non-critical in comparison. The runtime keeps track of the currently maximum criticality value through an atomic variable which is updated when TAOs are scheduled and completed. If the TAO is critical, a suitable core is found, and the TAO is scheduled in the corresponding work-stealing queue.

We have implemented two strategies for finding the suitable core, one where the runtime is aware of its heterogeneity and one where it remains unaware. We chose to implement the criticality-based scheduling in these two ways in order to understand the performance effects of introducing more awareness of the resource types into the system.

For the *aware* strategy, the critical TAOs are statically placed onto any big core (random) and non-critical TAOs onto any LITTLE core (random) based on user defines of the available resources. In the *unaware* strategy, whenever we encounter a critical TAO, the PTT is used to find the best performing core based on previous executions with corresponding resource width. Non-critical TAOs are simply scheduled on a random core. Although the overhead of the scheduling increases when searching the table, it is the most portable solution as the runtime does not need any information about the platform other than what is gathered at runtime.

3.2.2 Weight-based Scheduling

The Bias scheduling by Koufaty et al [18] was implemented by using a bias for each application that describes which core is preferred. An applications bias is estimated by analyzing the amount of stall cycles from running on the respective cores and represents how suited the application is to the different cores. The application bias is dynamically calculated and aids the already existing scheduler to choose the most suitable cores.

Our implementation of a weight-based scheduling is similar in the calculation of the weight or bias to that of the bias scheduling which is dynamically calculated at runtime. In our implementation however, we make use of the PTT to calculate a weight value for each TAO. The weight is calculated by dividing the execution time of a LITTLE core by the execution time of a big core. The calculation is performed every time a TAO is woken up in the commit-and-wakeup and the result is compared to a system wide threshold. If the weight value of a TAO is greater than the current threshold, it is an indication that this TAO gains a larger speedup by running on a big core than the other TAOs. Thus, if a task has a weight value greater than the threshold, it is scheduled on a random big core and if a TAO has a weight value less than the threshold, it is scheduled on a random LITTLE core. The threshold value is initially set to 1.5 and is then updated at every comparison with a weighted ratio of 1:6 to the old threshold value to represent the mean weight value of the system, $threshold = \frac{potentialweight+(oldthresholdvalue*6)}{7}$, where the potential weight is the potential speedup calculated for the comparison.

3.3 Task Molding

In addition to the scheduling extensions that focus on big/LITTLE placement, we propose an extension to resize and mold the TAOs widths. With the introduction of the PTT, we have knowledge of past executions, both in which core is the fastest and which resource width is the most suitable. Since the runtime already carries the notion of moldable tasks we can utilize this to make more intelligent scheduling decisions by changing the resource width. With this in mind, we implemented a molding mechanism that changes the resource width based on past execution times as well as the system load. This molding mechanism can be applied in isolation or together with the other scheduling extensions and is also implemented within the commit-and-wakeup mechanism.

We propose two policies for changing the resource width at runtime: load-based and history-based molding. The load-based molding was implemented to be able to benefit from extra resources when the system load is low. If the system load is lower than the available resources, TAOs can be molded into a larger size to better exploit the potential resources. If a DAG has a low degree of parallelism this is highly advantageous but if the system load is high however, the history-based molding is of more interest. The history-based molding was implemented to adjust the resource width of the TAOs to what was, in previous iterations, most suitable. Based on the core it compares trace-table entries to find the preferred resource width. Specifically, it looks within its cluster at the recorded execution times for each potential leader of each resource width. The width corresponding to the best performance compared to the amount of resources, becomes the new resource width. So for a new width to be set, the recorded execution time for that width$\times$ the width has to be lower than the current execution time. This way a wider resource width is chosen if the execution time is worth occupying more resources for. Changing the width according to history is beneficial for heterogeneous architectures where the preferred resource width of a TAO could be different between cores. Further, it could also be adaptive to interference and other types of non-architectural heterogeneities which might affect the desired resource width.

These molding policies can be used separately or together. In our implementations we use the molding policies hierarchically where the resource is adjusted to the load primarily but if the load is too high to make a justification for resizing for idle resources the history-based policy is used. This way, the system can adapt to loads as well as find the most optimal resource width.

4
Evaluation Method

This chapter presents the materials and methods used to evaluate our scheduling implementations. Section 4.1 explains the hardware used for the evaluation platform as well as the Linux installation necessary to make the platform compatible with the runtime. Section 4.2 explains the synthetic benchmarks used to evaluate the implementation. Lastly, Section 4.3 covers the metrics and the general testing-process used for the evaluations.

4.1 Evaluation Platform

The evaluation of the scheduling implementations was conducted on a HiKey960 development platform equipped with an octa-core Kirin 960 processor using an ARM big.LITTLE architecture [20]. This processor has four ARM Cortex-A73 and four Cortex-A53 cores with 3GB DDR4 SDRAM in an ARMv8 architecture node. In this processor, one L2 cache is shared among the big and the LITTLE cores respectively, while each core has private L1 caches.
The initial software support for the HiKey960 is a pre-installed Android Open Source Project (AOSP) which is a mobile/tablet-platform environment based on the android Common Kernel. In order to boot, run and develop on the development platform, the following materials were used:

- Power adapter of 8V-18V with 2000mA.
- USB Type-A to USB Type-C cable needed for interfacing with the fastboot environment.
- USB keyboard and mouse as well as a monitor with HDMI (optional).
- UART to USB @1.5V cable for debugging and interfacing with the board.

4.1.1 Linux Installation

In general, the installation process of an operating system on the HiKey960 is done by connecting the HiKey960 to a Linux host (via USB-C to USB-A cable), booting the HiKey960 into fastboot mode and flashing down the images. Fastboot mode can be activated using switches on the back of the board and is based on the Android SDK Platform Tools [21]. In the installation guide provided by Linaro [1], there is a script for flashing the correct files for the AOSP operating system (flash-all.sh) which requires prebuilt images of the AOSP. Installation of Linux follows a similar

[1]https://www.96boards.org/documentation/consumer/hikey960/installation/linux-fastboot.md.html Accessed:2018-02-17

guide but with added installation of Unified Extensible Firmware Interface (UEFI). The Linux installation process is described in Appendix A.

At the time of writing there is no official support for Linux on the HiKey960 board. The Linux distribution is under development and not yet finalized where Linaro is working on an Open Embedded Linux [22] under the Yocto Project [23]. There are a couple of snapshots available under the *morty* manifest branch which are continuously being updated by Linaro[2]. There are also Debian images for the original HiKey Processor which some have used for the HiKey960 as well[3].

When installing the latest release of UEFI images, we used the Open Embedded images [4] and out of simplicity of a complete system image, we choose to use the HiKey Debian images available for the Linux distribution[5]. These provided the largest set of included support such as a package installer, network manager, appropriate libraries etc. We did however have trouble getting HDMI to work with this distribution and we were lacking a proper platform management. Since the Debian image used was originally made for the homogeneous octa-core HiKey, the platform management problem resulted in somewhat uncontrollable effects of dynamic voltage frequency scaling (DVFS). Proper platform management was available in the open-embedded distribution and attempts to develop in this environment was made. Unfortunately, in order to run XiTAO, substantial alterations and additions to the snapshot-image were required which we limited ourselves not to explore in greater detail. The problem resulted in potentially unknown thermal throttling for the evaluation process. Thus, the evaluation was carried out with careful consideration to this by cooling down and powering off the evaluation board between test cases.

4.2 Evaluation Benchmarks

As discussed in the scope, our implementation had to be evaluated using suitable benchmarks. By using synthetic benchmarks with different kernels, we can stress various parts of the system and display the capabilities of the runtime and the scheduler. Although a real-world application would have been beneficial for evaluating the realistic overall performance, the use of synthetic benchmarks shows performance for bottleneck scenarios. In addition, it allows for repeatable experiments which provides easily comparable results.

To construct the synthetic benchmarks, we selected three kernels which exhibits streaming, computing and data reuse properties respectively. To analyze the kernels and evaluate specifically interesting scenarios, configurable DAGs were created. Three types of DAG benchmarks were created: single and parallel chains, fork-join pattern and randomized DAGs. The implementation of the single and parallel chains configurable DAG is explained in Section 4.2.1. The fork-join configurable DAG is explained in Section 4.2.2. In Section 4.2.3, we explain the generation of a

[2]The versions of the OE distribution can be found at:http://snapshots.linaro.org/reference-platform/embedded Accessed: 2018-02-05

[3]http://snapshots.linaro.org/96boards/hikey/linaro/debian/latest/ Accessed: 2018-02-16

[4]http://snapshots.linaro.org/reference-platform/embedded/morty/hikey960/100/rpb/ Accessed: 2018-02-16

[5]http://snapshots.linaro.org/96boards/hikey/linaro/debian/17/ Accessed: 2018-02-17

randomized DAG containing the kernels making up our main synthetic benchmark. Finally, the chosen kernels are then discussed and profiled in Section 4.2.4.

4.2.1 Single and Parallel Chains

The simplest benchmarks consisted of single and parallel chains of subsequent instances of TAOs. These benchmarks were mainly used to profile the kernels to find suitable configurations for achieving their expected behaviors and to observe the characteristics of the different kernels on the big and LITTLE cores. The single and parallel chains of dags were generated with the parameters: the size of the DAG, the working-set size of the TAOs and the resource hint of the TAOs. Each chain in the DAG has one dedicated memory location used for input and output of data for the TAOs, enabling reuse between TAOs, where the size of the allocated memory depends on the input parameters. Running several TAOs in parallel will show interference between TAOs. An illustration of the single and parallel chain benchmarks is shown in Figure 4.1.

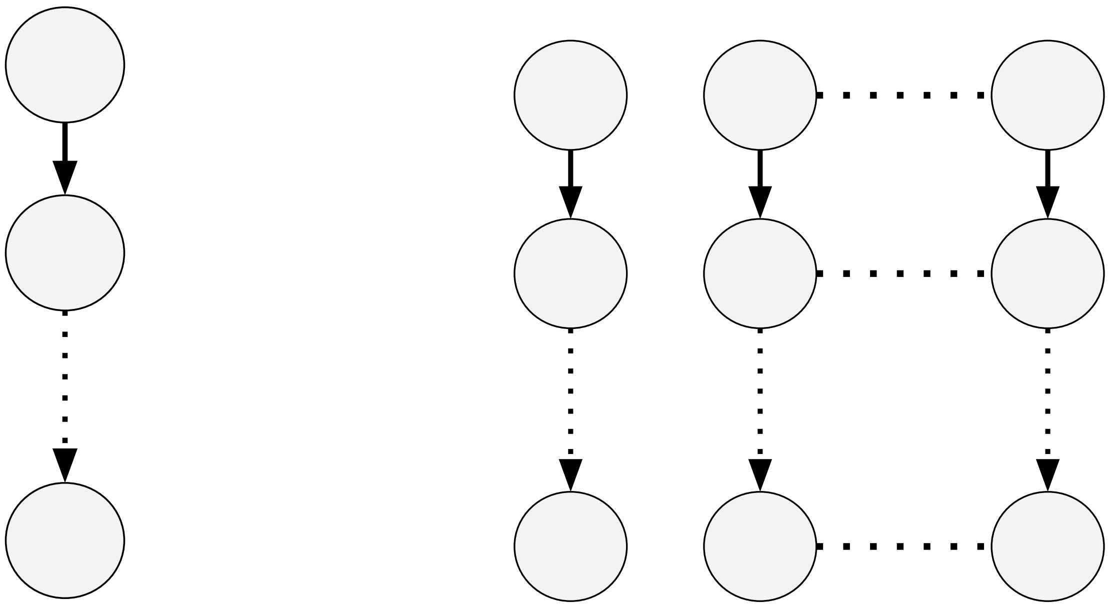


Figure 4.1: An example of the simplest evaluation benchmarks of repeated kernels, either in a single chain (left) or parallel chains (right). The single chain is generated from a desired DAG-depth while the parallel chains are generated from a desired DAG-depth and parallelism.

4.2.2 Fork-Join

The second benchmark we implemented was a configurable DAG with a binary fork-join pattern. The DAG starts with one node, forks like a binary tree until a desired parallelism (expressed as DAG-width) is reached and then shrinks until the DAG-parallelism is one again. If different types of TAOs are selected, they will be randomly distributed across the DAG. The memory is allocated so that each kernel reuses data from earlier levels with the same kernel. Figure 4.2 shows a DAG generated with 18 TAOs and a parallelism of four. This DAG can be used to test how the scheduler performs when the degree of parallelism rapidly changes.

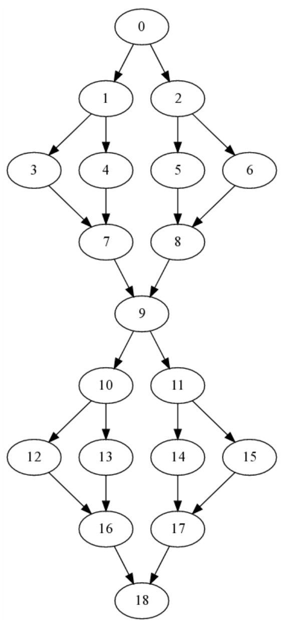


Figure 4.2: An example of a fork-join DAG with a maximum parallelism of four. The numbers within the nodes are the node numbers.

4.2.3 Randomized DAG

While the benchmarks that highlight specific scenarios are interesting, to properly evaluate the performance of the scheduler, a random composition of the kernels were implemented. A random composition can effectively imitate how the behavior of an application keeps changing.

To generate a suitable randomized DAG, a set of configuration parameters were used, similar to the generation of DAGs by Topcuoglu et al. in [2]. The first parameter is the number of TAOs of each kernel, this can be used to choose which kernel should be most prominent in the DAG. The second parameter is the size of the TAOs, this can be used to choose the amount of workload per TAO. Additionally, the average width of the DAG can be adjusted to obtain the desired parallelism. The last configuration parameter, the edge rate parameter, decides the average amount of connected edges a TAO will have, this will also affect the parallelism of the DAG. A seed value is used to manipulate the randomization to recreate a DAG several times for comparison.

The DAG generation algorithm produces a DAG in three steps. The first step generates the shape (nodes and edges) of the DAG. This step is separated from the TAO creation step in order to get proper memory utilization and data reuse.

The second step consists of allocating memory and deciding which TAOs are reusing data. To achieve data reuse between nodes, we maintain a vector for every kernel where each index in the vector represents a memory location. Initially, the size of the vector is zero. For every node, we search its predecessors for a node number matching any of the numbers in the vector. If a matching number is found, it is replaced by our current node number and the index to the location is saved in the node. If we cannot find a match, a new entry is created in the vector with the unique node number and that index is saved in the node instead. The size of the vectors is then used for allocation of memory and each node will have a designated data location. The memory is allocated this way to maximize data reuse between TAOs of the same kernel while guaranteeing isolated data execution when TAOs are run in parallel.

The final step is to traverse the nodes and spawn the corresponding TAOs and edges between them. The DAG generator also prints an output file in the graph description language DOT which is used to visualize the graphs. An example of a generated DAG can be seen in Figure 4.3.

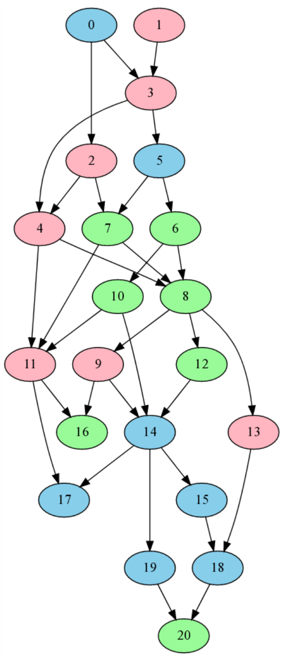


Figure 4.3: An example of a generated random DAG. The red nodes represent matrix multiplication TAOs, the blue nodes represent sort TAOs and the green nodes represent copy TAOs. The numbers within the nodes are the node numbers.

4.2.4 Kernels

When selecting our kernels, the priority was to match the desired characteristics and, preferably, without requiring considerable implementation effort. The chosen kernels were: a copy kernel, a sorting kernel and a matrix multiplication kernel.

A *copy* kernel handling large inputs was implemented for the streaming property. The kernel needs to read and write to memory frequently, effectively creating a streaming behavior where the kernel has to access the main memory continuously.

For the data reuse property, a *quicksort* and *mergesort* kernel combination was chosen. This kernel first splits the input array into chunks and performs in-place sorting with quicksort before carrying out two levels of merge sorting, effectively reusing the data within the kernel.

Finally, a *matrix multiplication* kernel was created for the compute-intensive property. We implemented a matrix multiplication that can benefit greatly from parallelism by ensuring that the writing of output data was done to separate cache lines for each thread while still sharing input data.

The copy and matrix multiplication kernels were implemented for this thesis specifically while the quicksort and mergesort sequence was created for a parallel sort benchmark used by Pericàs in [5]. These kernels were chosen for simplicity and availability. In order to minimize the potential errors while providing the correct characteristics, simple kernels were preferred.

Kernel Profiling

To understand the behavior of the chosen kernels, we profiled them on the evaluation platform with the XiTAO runtime. As previously mentioned, we made use of the configurable DAGs with single and parallel chains of TAOs for this purpose. We measured throughput (TAOs/s) of each kernel for different DAG configurations and resource hints. The DAG configurations with parallel chains could highlight potential interference in the kernels such as memory congestion etc. In addition, different resource hints allowed us to see how the kernels responded to having multiple cores share the workload of a single TAO. The profiling also allowed us to see potential speedups and behaviors of the kernels on big and LITTLE cores respectively.

For each kernel, we chose the appropriate working set size corresponding to the desired behavior. For the matrix multiplication, we chose a 64x64 matrix multiplication, as it handled the largest number of computations per second. For the sorting, we chose a 262kB input array, taking up a total space of 524kB, effectively fitting in the L2 caches of the big and LITTLE cores. Finally, the copy used a 16.8MB array, taking up a total space of 33.6MB, exceeding the space of the L2 caches. These working set sizes also resulted in similar execution times for the kernels on the LITTLE cores. This is suitable for the benchmarks as it makes the task granularity equal across our different TAOs and it becomes simpler to monitor the fraction of the execution time for each kernel.

We adjusted the length of the configurable DAGs to 1000 repeated TAOs to get a reasonable amount of test data. For each profile configuration, we ran five measurements with an interval of ten seconds. The evaluation board was cooled down for five minutes between each configuration, this was done to counteract any potential DVFS or other throttling. The results of the profiling can be seen in Figure 4.4 for the matrix multiplication, Figure 4.5 for the sort and Figure 4.6 for the copy.

The profiling of the matrix multiplication kernel shows a linear increase in the throughput with the number of cores, both in the case of one TAO using multiple resources and running multiple TAOs in parallel. The speedup of big over LITTLE cores is approximately 2.4 in all of the configurations.

For the sort kernel, the profiling shows a higher throughput if there are multiple TAOs rather than multiple resources working on the same TAO. The internal dependencies within this kernel translates into different degrees of parallelism in different stages, therefore limiting the potential performance increase for some stages, which could be the reason for the slight decrease in performance in the larger resource hints. As allocated memory is effectively 524kB multiplied by the number of parallel chains, the performance of the 2×1 and 4×1 configurations suffers from interference in the shared cache as the working set no longer completely fits. The sorting only performs marginally better on the big cores rather than the LITTLE cores.

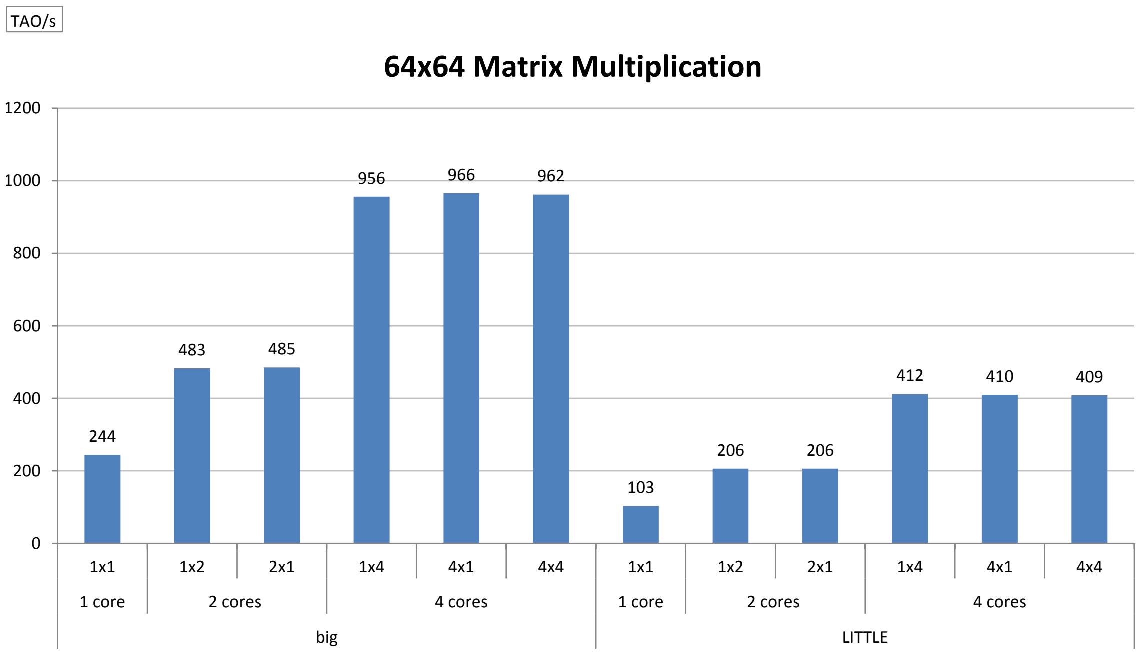


Figure 4.4: Profile of matrix kernel showing resource width and parallel chain combinations on both big and LITTLE cores. M×N denotes the number of parallel chains × the resource hint. For example, 1×1 denotes one TAO chain on one core while 1×2 denotes one TAO chain on two cores. Each resulting throughput value (TAO/s) is the mean of the five measurements taken on that configuration.

Finally, the profiling of the copy kernel shows limited performance gains from increased parallelism, especially when running on the big cores. This is the effect of reaching the memory bandwidth limit due to continuously accessing the main memory. The copy kernel performs significantly better on big cores rather than LITTLE cores which could be due to architectural differences between the cores as the big cores are out-of-order superscalar cores with a maximum throughput of two instructions per cycle and the LITTLE cores are 2-way superscalar in-order cores.

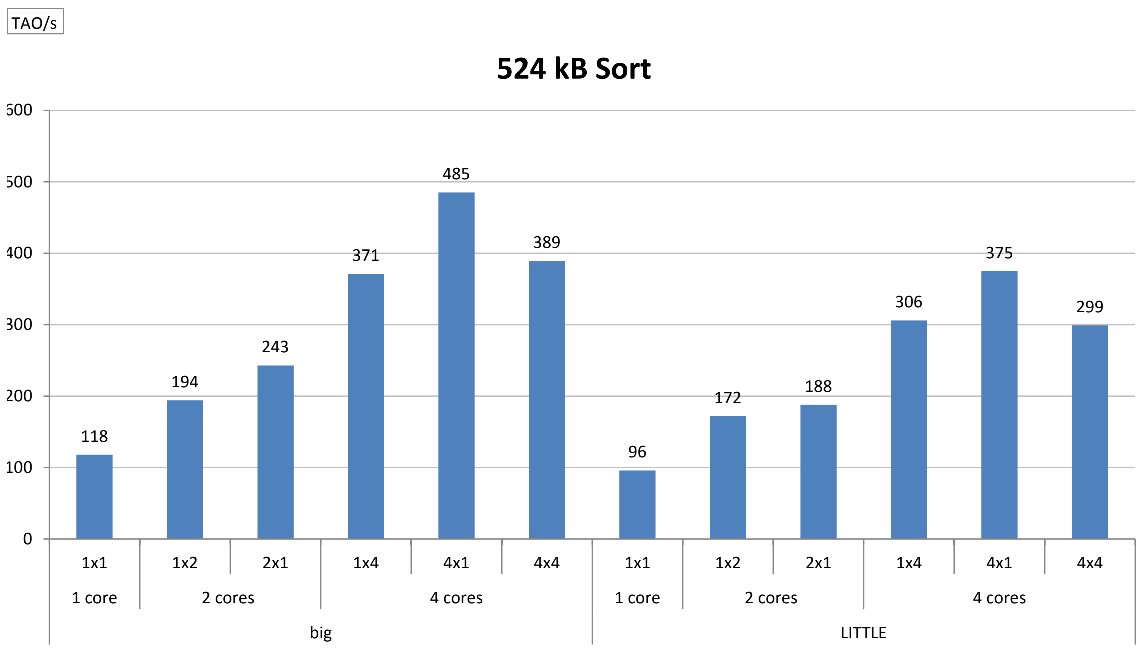


Figure 4.5: Profile of sorting kernel showing resource width and parallel chain combinations on both big and LITTLE cores. M×N denotes the number of parallel chains × the resource hint. For example, 1×1 denotes one TAO chain on one core while 1×2 denotes one TAO chain on two cores. Each resulting throughput value (TAO/s) is the mean of the five measurements taken on that configuration.

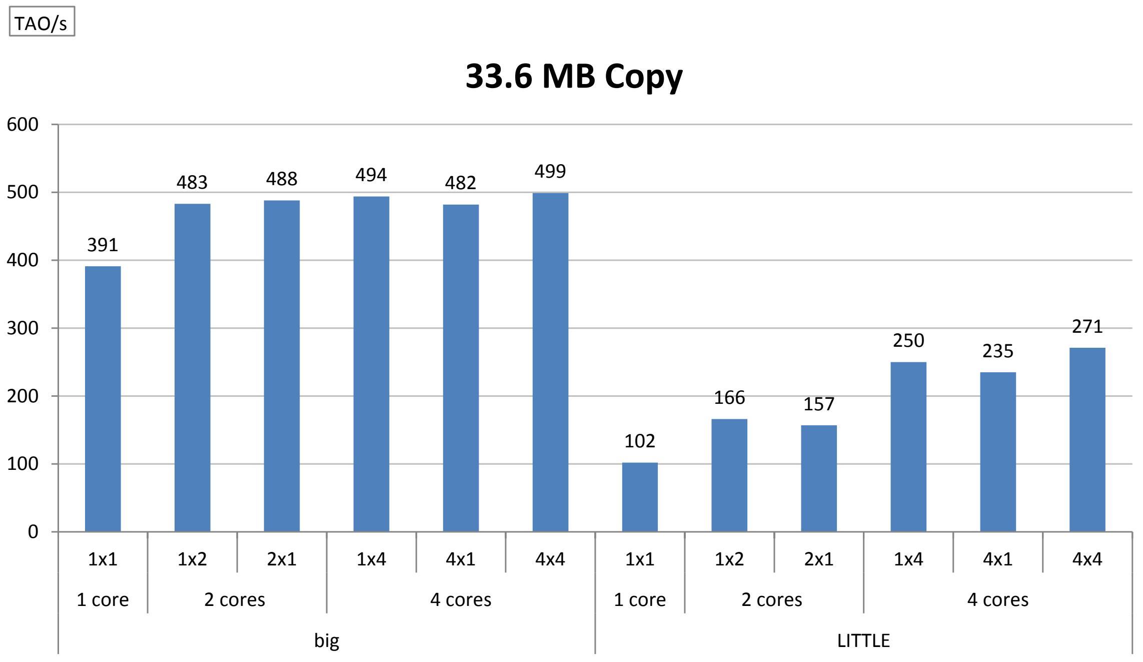


Figure 4.6: Profile of copy kernel showing resource width and parallel chain combinations on both big and LITTLE cores. M×N denotes the number of parallel chains × the resource hint. For example 1×1, denotes one TAO chain on one core while 1×2 denotes one TAO chain on two cores. Each resulting throughput value (TAO/s) is the mean of the five measurements taken on that configuration.

4.3 Evaluation Metrics and Methodology

In order to evaluate the integrated scheduling extensions we chose to compare it to the original runtime which makes resource aware-scheduling without considerations of heterogeneity. This allows us to effectively compare the performance impact of prioritizing TAOs based on the available heterogeneity to a case where it is not considered. Our evaluation metric for this is the throughput, TAOs/s, as it allows us to measure how effective the placement of the TAOs have been in terms of performance. Other metrics such as energy-delay-product would have been interesting to consider as well, but due to lack of support on the evaluation platform (platform management, lacking counters of correlating metrics etc.), we chose a readily-available performance metric.

For the evaluation methodology, we produced results on the Hikey960 board with the Debian OS using our evaluation benchmarks. In order to present comparable results, each test case was performed on a cooled down and recently booted board following a minimum break of seven minutes with the board turned off. This was to counteract potentially skewed results due to thermal throttling. Each test case was run in five consecutive runs with an interval of ten seconds and the presented evaluation metric TAOs/s is the arithmetic mean value of the gathered measurements. Diverging values due to throttling is then equally accounted for in all test cases.

The degree of parallelism for the evaluation benchmarks is another important metric to consider when analyzing the results of the scheduling implementations. We define the degree of parallelism as $\#TAOs/Cp$ where Cp denotes the length of the critical path. This definition allows for a simple perspective of the parallelism of a DAG.

5
Results

This chapter presents the results for the evaluation of our scheduling extensions presented in Chapter 3. First a description of the testing environment, notations and the configuration parameters are given. The results for the benchmarks are then presented in Section 5.1, 5.2 and 5.3. Finally, additional results of the task molding extension is presented in Section 5.4.

The evaluated scheduling extensions are denoted: Criticality-based scheduling (heterogeneity aware) for the implementation using defines of which cores are big and LITTLE, Criticality-based scheduling (PTT) for the implementation that is unaware of any heterogeneity other than what is discovered at runtime and Weight-based scheduling for the implementation that selects core based on performance gains comparisons. The molding of the resource width is used together with the criticality-based (PTT) scheduling and the weight-based scheduling. For the task molding, the weight-based and the history-based policies are used together. The base case in these evaluations is the runtime without any of the scheduling extensions, we denote this the homogeneous scheduling.

All evaluations were tested with the resource hint set to one and four in width. The working set sizes for the kernels are the profiled scenarios shown in Section 4.2.4, 64×64 for matrix multiplication, 524kB for the sort and 33.6MB for the copy. The general parameters used in the XiTAO runtime are listed in Table 5.1. The evaluation was done using the methodology described in Section 4.3.

Table 5.1: The XiTAO configuration parameters used in the evaluations. The number of threads equals the number of cores. The load balancing policy is work stealing where the thread attempts to steal one time before retrying the `worker_loop`. The runtime is compiled with compilation flag -O0. This was to get the desired behaviors of our kernels.

Configuration Parameter	**Value**
Threads	8
Load balancing policy	Work stealing
Steal attempts	1
Optimization flag	-O0

5.1 Parallel Chains Evaluation

We ran two test cases with parallel chains of TAOs. The first test case consisted of three chains, each 1000 TAOs, and the second consisted of twelve chains, each 200 TAOs. The amount of TAOs for each of our kernels were equal, one chain/kernel and four chains/kernel respectively. The results of the two test cases are presented in Figure 5.1 and 5.2 respectively.

For the three parallel chains, the scheduling extensions utilizing the molding of tasks perform similarly with an approximate speedup of 2.67 over the homogeneous scheduling base case with resource hint four. The heterogeneous-aware criticality scheduling without molding, performs approximately the same as the other scheduling extensions for resource width four. For resource width one however, it shows a speedup of 1.18 while the extensions with molding show an approximate speedup of 2.82 over the homogeneous scheduling. The homogeneous scheduling performed similarly with resource width one and four which is mainly due to the LITTLE cores executing all of the chains. This is because the homogeneous scheduling forwards TAOs if possible. If the TAOs are placed on the LITTLE cores initially the homogeneous scheduling forwards the TAOs locally without allowing the big cores a chance to steal. Utilizing the STA property to get better placing could result in better throughput for the homogeneous scheduling.

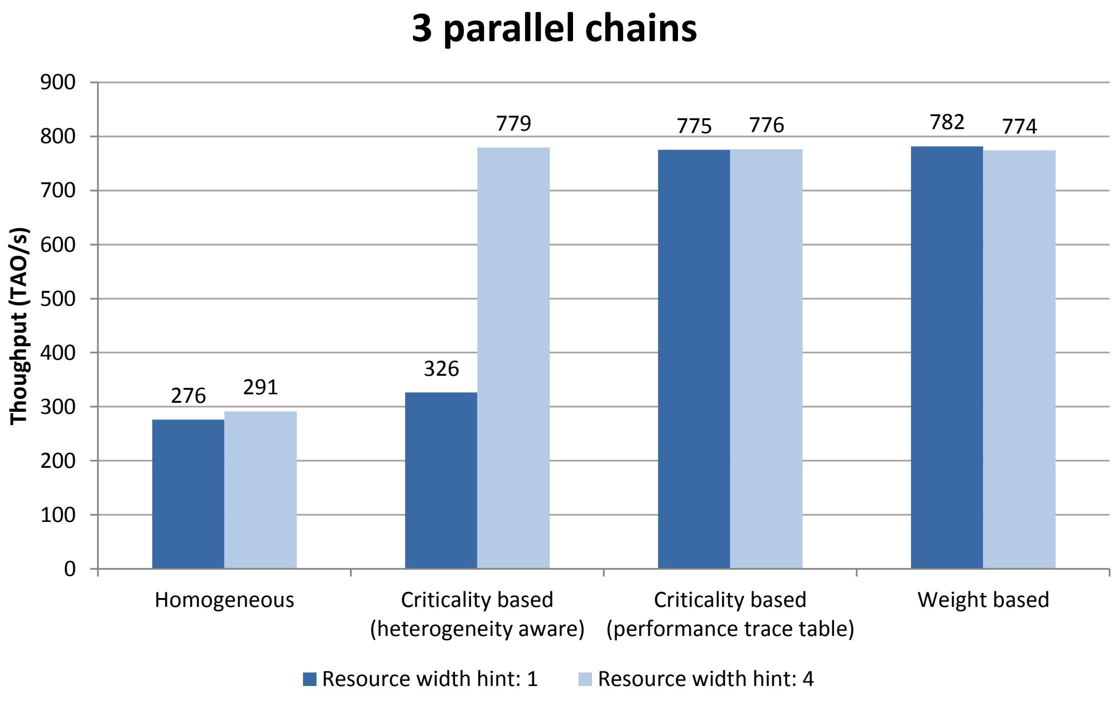


Figure 5.1: Resulting throughput values for the evaluated scheduling extensions for a three-chain benchmark with resource hint one and four. The criticality-based (performance trace table) as well as the weight-based extensions utilize task molding.

For the twelve parallel chains the overall improvement is less than for the three parallel chains due to the increased parallelism. The speedup is approximately 1.52

for the heterogeneity-aware criticality scheduling with resource width one. The criticality-based (PTT) and weight-based scheduling which uses task molding, show a speedup of 1.55 and 1.62 respectively over the homogeneous scheduling with resource width one. When using a resource width of four, the heterogeneity-aware criticality scheduling has a speedup of 1.09 while the scheduling extensions with molding have an approximate speedup of 1.45.

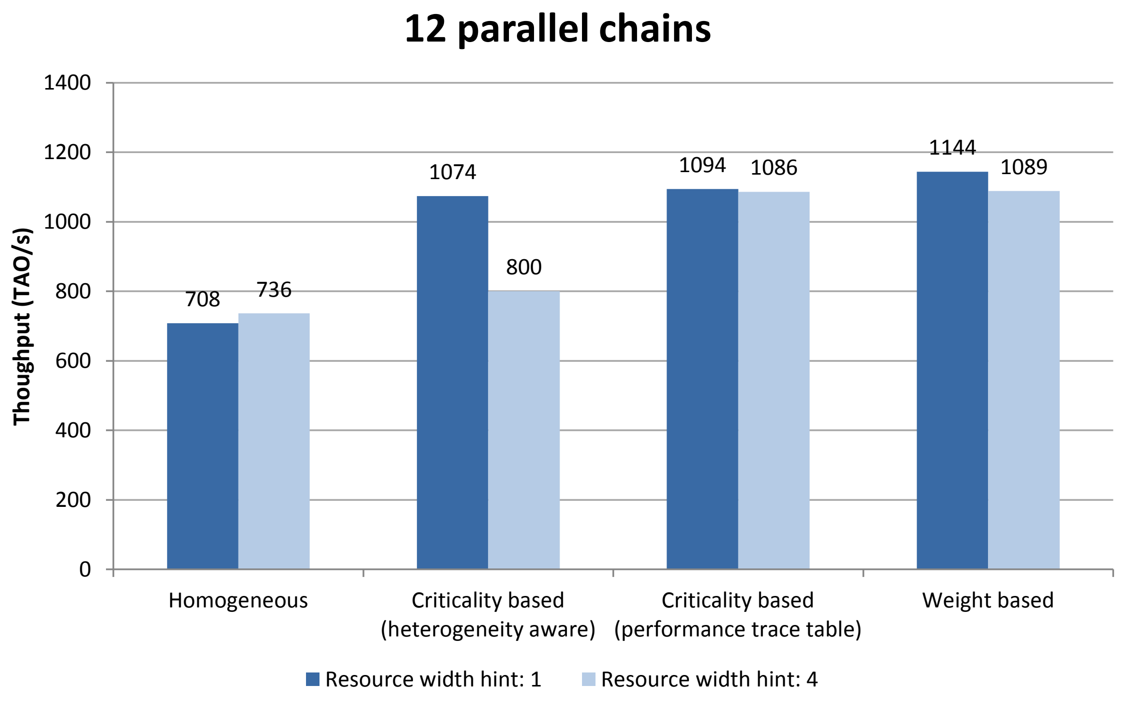


Figure 5.2: Resulting throughput values of the scheduling extensions for a twelve-chain benchmark with resource hint one and four. The criticality-based (performance trace table) as well as the weight-based extensions utilize task molding.

5.2 Fork-Join Evaluation

We ran one test case with the fork-join DAG where the DAG was generated with a maximum DAG-width of 8 and 1000 TAOs of each kernel. The results of this evaluation can be seen in Figure 5.3. A similar behavior as the parallel chains can be observed in regards to with and without the task molding. The homogeneous scheduling with a resource hint of four is beneficial for this DAG as width one leaves resources idle where the DAG has a low degree of parallelism. The approximated speedup of the scheduling extensions with molding is 1.24 over the homogeneous scheduling with resource width four and 2.08 over resource width one. The speedup of the heterogeneity-aware criticality scheduling is 1.19 over the homogeneous scheduling with a resource width of four and 1.21 over resource width one.

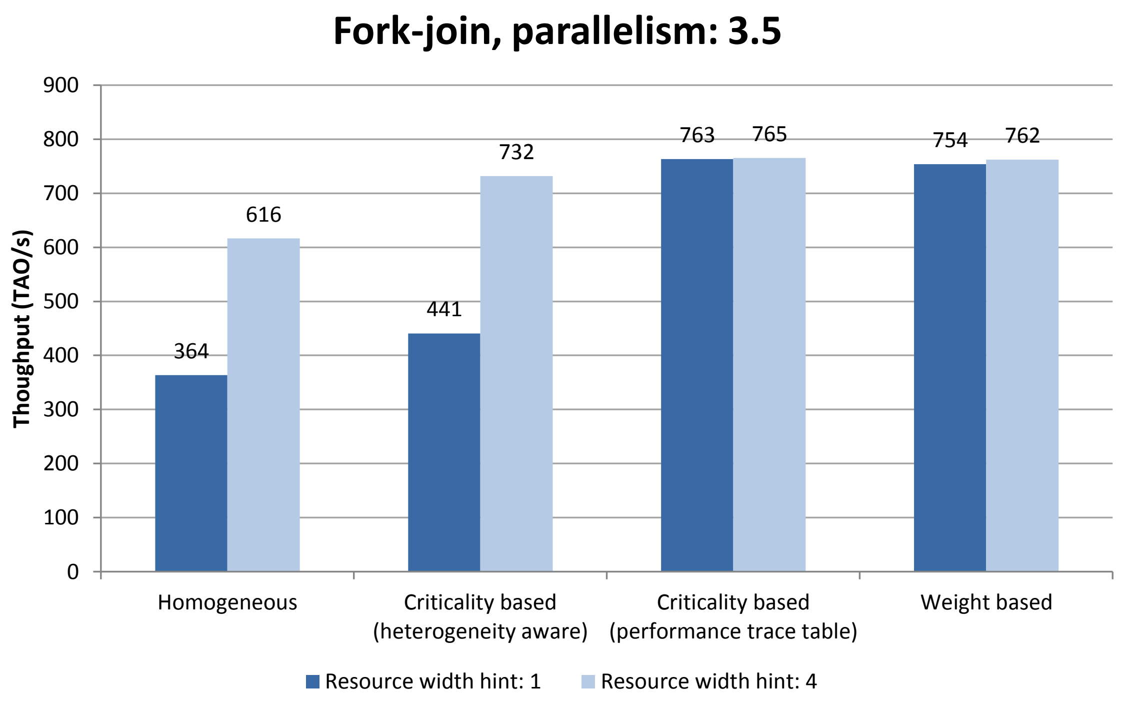


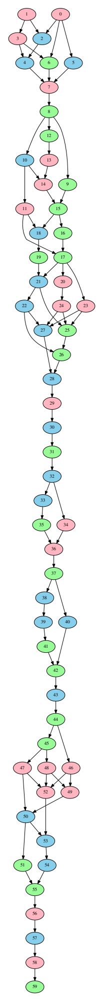
Figure 5.3: Resulting throughput values of the scheduling extensions for a fork-join benchmark with a degree of parallelism of 3.5 for resource hint one and four. The criticality-based (performance trace table) as well as the weight-based extensions utilize task molding.

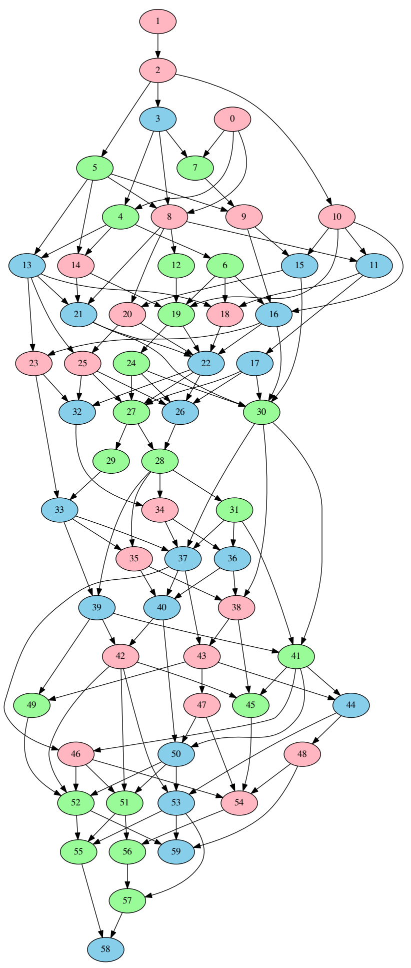
5.3 Randomized DAG Evaluation

We generated three randomized DAGs with different degrees of parallelism: 1.62, 3.03 and 8.06. Each DAG consisted of 3000 TAOs and representations of the first 60 nodes in the DAGs are shown in Figure 5.4. The figures show how a section of the DAGs might look in terms of structure and parallelism. The configuration parameters listed in Table 5.2 are the parameters used to create these DAGs.

Table 5.2: Random DAG generation parameters for the three generated DAGs.

Degree of Parallelism	Edge Rate	Level Width	Seed
1.62	55	1	123
3.03	32	2	123
8.06	8	2	123


(a) Representing degree of parallelism of 1.62.


(b) Representing degree of parallelism of 3.03.

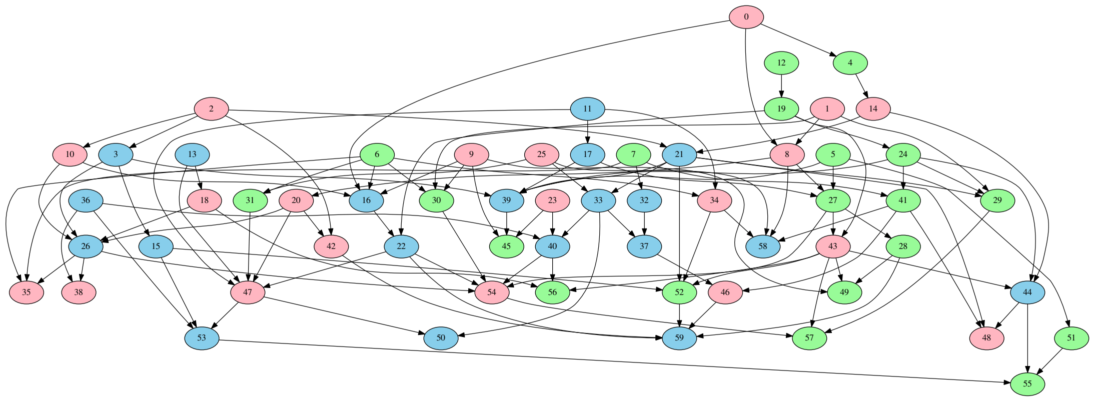

(c) Representing degree of parallelism of 8.06.

Figure 5.4: Respective figure is scaled for aesthetics. The figures show scaled down versions of the randomized DAGs used for the respective evaluations. The original number of TAOs are 1000/kernel and the scaled down version showed here are 20/kernel. Please note that the degree of parallelism in these DAGs may not perfectly correspond to the original version but their structures are the same. The numbers within the nodes are the node numbers.

The results of the DAG with a degree of parallelism of 1.62 can be seen in Figure 5.5. The approximate speedup of the scheduling extensions with molding is 1.29 over the homogeneous scheduling with width four and 2.78 over width one. The heterogeneity-aware criticality scheduling with resource width four shows the largest speedup, 1.31, over the corresponding homogeneous base case. The heterogeneity-aware criticality scheduling shows 1.19 speedup over the homogeneous scheduling with width one.

The results of the DAG with a degree of parallelism of 3.03 can be seen in Figure 5.6. A similar behavior to the fork-join can be seen in these results. The approximate speedup of the scheduling extensions with molding is 1.27 over the homogeneous scheduling with resource width four and 2.03 over width one. In comparison to the DAG with parallel degree of 1.62, the speedup over the width one homogeneous case is less. For the heterogeneity-aware criticality scheduling, width four shows a similar speedup with 1.28 over the corresponding homogeneous scheduling base case. Similarly to the DAG with a parallelism degree of 1.62, the heterogeneity-aware criticality scheduling with width one shows a smaller speedup.

The results of the DAG with a degree of parallelism of 8.06 can be seen in Figure 5.7. For this benchmark, resource width one shows the best throughput for the homogeneous scheduling. The approximate speedup of the scheduling extension with molding, is 1.1 over the homogeneous scheduling with width one and 1.28 over width four. For the heterogeneity-aware criticality scheduling, width four shows a similar speedup to the scheduling extensions with molding, with 1.07 over the corresponding homogeneous scheduling base case. For this benchmark, width four instead shows a smaller speedup for the heterogeneity-aware criticality scheduling.

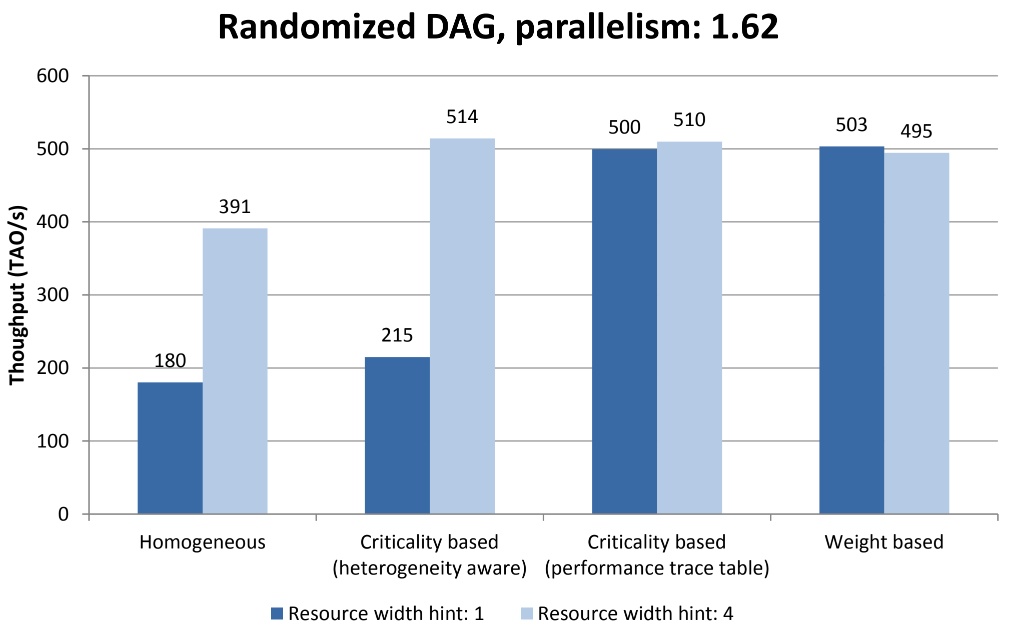


Figure 5.5: Resulting throughput values of the scheduling extensions for a random-generated DAG benchmark with a degree of parallelism of 1.62 for resource hint one and four. The criticality-based (performance trace table) as well as the weight-based extensions utilize task molding.

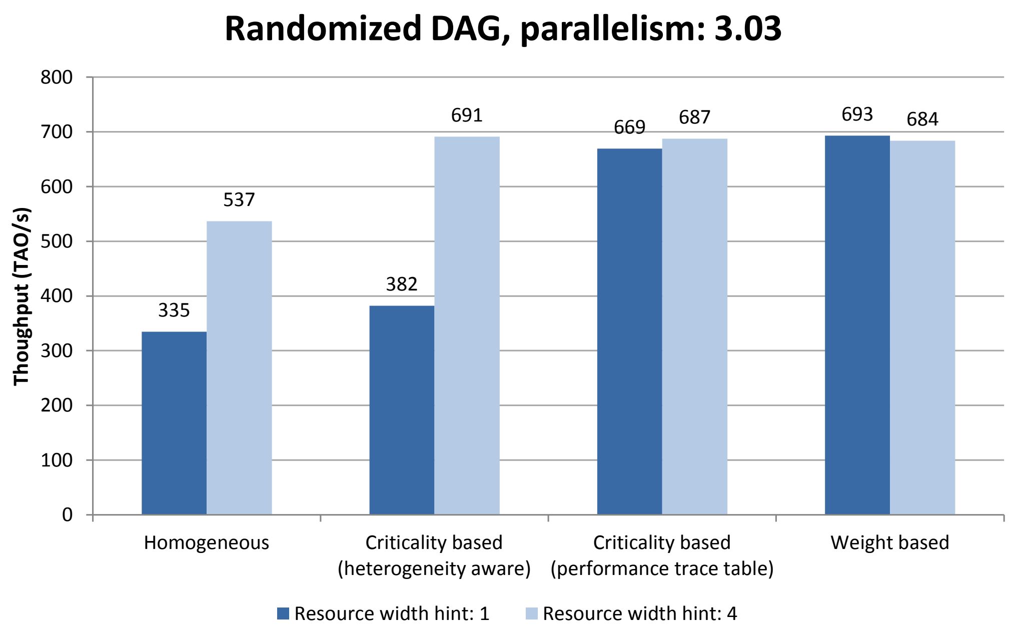


Figure 5.6: Resulting throughput values of the scheduling extensions for a random-generated DAG benchmark with a degree of parallelism of 3.03 for resource hint one and four. The criticality-based (performance trace table) as well as the weight-based extensions utilize task molding.

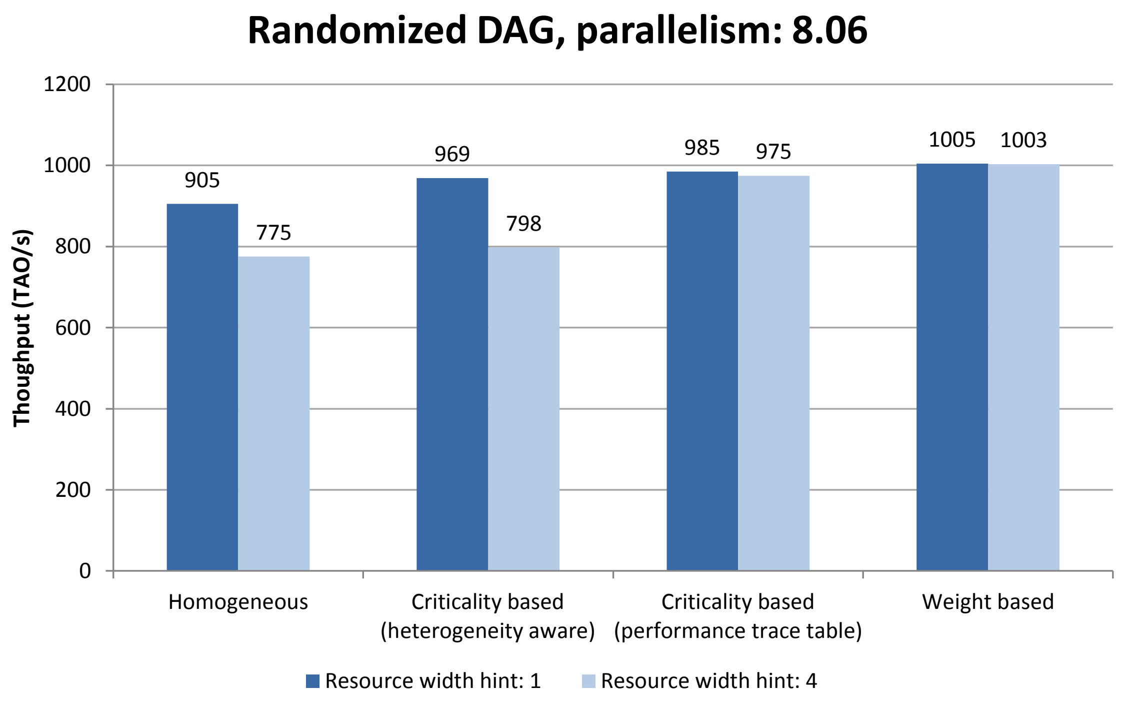


Figure 5.7: Resulting throughput values of the scheduling extensions for a random-generated DAG benchmark with a degree of parallelism of 8.06 for resource hint one and four. The criticality-based (performance trace table) as well as the weight-based extensions utilize task molding.

5.4 Task Molding Evaluation

In Table 5.3 and Table 5.4 the throughput values for the three randomized DAGs are presented with the two respective scheduling extensions, criticality-based scheduling (PTT) and weight-based scheduling, with and without molding. Small overheads of using molding together with the two scheduling extensions can be seen in the case of 1.62 or 3.03 degree of parallelism, where at most the weight-based scheduling without molding achieves a 1.01 speedup over the corresponding case with molding. In the heavily parallel DAG the molding of the resources instead show a performance gain over the case without molding, at most the speedup is 1.08 with the criticality-based scheduling.

Table 5.3: Throughput of weight-based scheduling with and without molding for the three randomized DAGs. The resource hints corresponds to the resource hints resulting in highest throughput for the base case homogeneous scheduling.

DAG(Degree of Parallelism)	Without Molding	With Molding
1.62 (Resource Hint = 4)	497	495
3.03 (Resource Hint = 4)	693	684
8.06 (Resource Hint = 1)	946	1005

Table 5.4: Throughput of criticality-based scheduling (performance trace table) with and without molding for the three randomized DAGs. The resource hints corresponds to the resource hints resulting in highest throughput for the base case homogeneous scheduling.

DAG(Degree of Parallelism)	Without Molding	With Molding
1.62 (Resource Hint = 4)	510	510
3.03 (Resource Hint = 4)	695	687
8.06 (Resource Hint = 1)	909	985

6
Discussion

This chapter aims to provide insight to the results found in Chapter 5 as well as to analyze and evaluate our design choices and evaluation methodology described in Chapter 3 and Chapter 4 respectively. A discussion of the results is first given in Section 6.1, the evaluation environment is then discussed in Section 6.2 and finally, a discussion regarding our design choices, their flaws and future suggestions, is given in Section 6.3.

6.1 Results

From the results in Chapter 5 we have shown that there are plenty of opportunities to make significant performance gains by making the scheduling resource aware for a heterogeneous platform. The speedups from the benchmark evaluations ranged from 1.1 to 1.45 in comparison to the homogeneous scheduling with an appropriate resource hint. Overall, the results show smaller speedups for the scheduling extensions on a DAG with a higher degree of parallelism because the homogeneous scheduling can utilize all cores more efficiently since there are always tasks to execute. Although larger speedups can be seen when the resource hint is poorly chosen, the scenarios when it is well chosen show the scheduling extensions' ability to choose the correct resource. However, it is clear that all our scheduling extensions result in performance increases over the homogeneous scheduler in all evaluations.

The results of the scheduling extensions on randomized DAGs were compared with and without the molding in order to see if the task molding is beneficial when the resource hint is well chosen. Here it becomes clear that depending on the type of DAG, performance can decrease, increase or be equivalent when scheduling with or without molding. For a lower average degree of parallelism the resulting throughput can be somewhat better without molding. This can potentially be due to the DAG having a low enough degree of parallelism for it to be more important to only consider the resource capability while keeping the resource width high. The additional overhead of task molding might cancel out the benefits if the ideal resource width is given as a hint. However, the molding shows significant performance increase for the more parallel case compared to without molding.

Furthermore, we also get interesting results for the scheduling extensions using task molding in comparison to the homogeneous scheduling where the resource hint is **not** appropriate for the DAG. These cases show speedups ranging from 1.28 to 2.82 where finding the optimal resource width seemingly pays off. These improvements combined with the speedup gained in more parallel DAGs is enough to diminish the overheads of the molding.

Overall, the least parallel DAG shows the most improvement of heterogeneous scheduling as it effectively utilizes the big cores. Additionally, there is little to no difference between the criticality-based scheduling (PTT) as well as the weight-based scheduling in any of the evaluated cases. The weight-based scheduling shows perhaps slightly better performance in the evaluations with a higher degree of parallelism, whereas the criticality-based scheduling somewhat exceeds in the opposite cases.

6.2 Evaluation Environment

As we have previously mentioned, we chose to evaluate the scheduler with the help of synthetic benchmarks in order to create comparable scenarios with interesting loads and behaviors. The kernels chosen for these evaluations were Matrix Multiplication, Sort and Copy in order to evaluate three different behaviors. The profiling of the kernels showed that the three kernels indeed had the desired behaviors, however, it would also have been interesting to include a larger variety of kernels in order to create evaluation benchmarks even more representative of reality. Similarly, it would have been interesting to consider more evaluation benchmarks. Ideally, we would want to evaluate the scheduling extensions with real applications as the next step.

For all the benchmarks, we have chosen to compare our scheduling extensions to the runtime without any scheduling extension, the homogeneous scheduling. Although a base case or a comparison to other heterogeneous scheduling would have been interesting to understand the performance gains of our specific extension, the aim of this thesis was to improve the runtime scheduling to consider heterogeneous platforms. We also chose to compile with a weaker optimization flag in order for the kernels to maintain the desired behavior which affects the compilation of the runtime itself, cross-compilation on other platforms with other kernels and higher optimization flags would be of most interest in order to confirm the results of our thesis. We do however only evaluate the scheduling extensions in comparison to our chosen base case which is compiled identically and the results are therefore still relevant.

The evaluation platform used in this thesis was the Hikey960 which proved to be a somewhat troublesome platform to use mainly due to the limited support. Although the Linux installation process was simple and the material readily available, the limited support was discovered at a later stage. Effects of thermal throttling was not visible in any testing until later in the profiling stage and although attempts were made to improve the platform it could not be fixed with our limited time. The effects of this were mainly prolonged testing, limited evaluation metrics and variable data points. The limited evaluation metrics affected the focus of thesis towards only considering performance. While performance in terms of throughput is interesting, other evaluation metrics enabling us to consider the energy savings or energy delay product to a further extent would have been desirable.

To counteract any variable data points, the data from our evaluations are taken from a specific testing process in order to present accurate and reliable data. It is however, in the interest of the scheduler to be able to manage heterogeneous behaviors such as throttling due to overheating which we consider a natural type of heterogeneous behavior, not a limitation. Furthermore, the weight-based scheduling which bases the weights of the TAOs on the execution history would probably excel in these scenarios as it can detect changes to the system.

Ideally, the results of this thesis should be cross-referenced with other platforms. This would allow for further evaluation of our schedulers and allow for energy metrics to be properly considered.

6.3 Design Choices

The design choices for this thesis are explained and discussed to some extent in Chapter 3. In this section we want to emphasize specific implementation suggestions for future work while discussing potential flaws in the presented scheduling extensions.

For all the scheduling extensions, we made the decision to implement them in the same commit-and-wakeup routine in order to preserve the underlying mechanisms of the runtime. This allows our implementations to choose any available resource while the DPA and the work-stealing will effectively distribute the actual tasks at a later stage. Naturally, the scheduling decision can be moved and perhaps be extended to make more intelligent or fine-grained decisions. It might be of interest to consider a more fine-grained scheduling which could result in fewer possible steals, but this would be a trade off with the overhead of introducing more scheduling decisions. In addition, extending the scheduler to consider STA values would also be an interesting implementation for future work.

Alternations to the implementations of the scheduling extensions could naturally be explored. Specifically, the criticality-based implementations could consider analyzing sections of the DAG with data from the PTT rather than the entire DAG at the start of execution. This might provide a more accurate depiction of the critical path, especially if the execution time of the different TAOs are significantly varied. In addition, it could also perhaps allow for a more dynamic solution which does not rely on a static analysis of the DAG structure done prior to execution.

As briefly mentioned, more intelligent work stealing was considered with the scheduling extensions. One idea was to not allow critical TAOs to be stolen or stolen by a LITTLE core. Another was directed stealing where LITTLE cores first try to steal from only LITTLE cores and big cores from only big cores. This proved inefficient compared to the current random work-stealing at an earlier testing stage. Further consideration to the work stealing together with the scheduling extensions can be considered for future research.

In order to make intelligent scheduling decisions based on the available resources at runtime, we used the PTT to record execution time for the different core-width pairs. For this thesis, execution times in seconds was readily available and cross-platform portable. We have considered other metrics such as stall cycles or cache misses, effectively allowing the scheduling to target other performance metrics depending on the goal of the scheduler but decided to use execution time as it was readily available despite the limitations of our platform. Targeting other metrics, especially energy related metrics is one of the most interesting directions for future work.

Furthermore, the PTT was implemented to be able to trace the execution times of the different TAOs. A downside of this is the programming overhead as it requires explicit declarations of the static table and the corresponding get and set functions, for each TAO class. While this is not the most ideal solution, the runtime is intended to be used as a backend to other models and not manually programmed through the programming API. We have also stated that the overhead recording the performance of the TAOs is less than 1%, while this is true if the TAOs are of the current size, this overhead might become significant if the TAOs are smaller. The table would also scale with the number of cores, which would make reading and comparing values less effective if not careful.

It should be noted that the criticality-based scheduling (performance-trace-table) is the most portable solution as it actually discovers the heterogeneity at run time. While this might be desirable it comes with the added overhead of always having to search for the most suitable resource. The criticality-based scheduling (heterogeneity aware) is then interesting if we assume that the resource hint is well-chosen as it does not need to trace history to make intelligent decisions. The awareness requires knowledge of the platform prior to execution but it is the simplest scheduling extension, effectively minimizing overheads. This is also the reason to why this scheduling extension did not use the task molding as it would add overhead of finding the resource width. Any of the criticality-based scheduling extensions does require an analysis of the input DAG prior to execution which can be limiting the performance if the DAG allows for TAOs to be created at runtime. The weight-based scheduling can easily be extended with other thresholds to target for example, energy-delay-product, by only scheduling TAOs that gain **enough** speedup on big cores. It is, similarly to the criticality-based (heterogeneity aware) scheduling, dependent on knowledge of the platform prior to execution which could perhaps be implemented to be discovered at runtime for a more portable solution.

Additionally, the scheduling extensions which require platform knowledge are currently confined to the architecture within our scope, a big.LITTLE architecture. Since big.LITTLE architectures are commonly used today this is a reasonable limitation but it would however be interesting to see how the scheduling extensions perform on architectures with various capabilities but without the notion of core clusters. The criticality-based scheduling (heterogeneity aware) and the weight-based scheduling could then be modified to find and create core clusters either dynamically at runtime or as a preface routine. A dynamic creation of core clusters, allowing for changes to the clusters during execution, could be interesting to account for other heterogeneous behaviors such as thermal throttling etc.

Finally, there are also some potential flaws in the task molding implementation which could be improved for even better performance. For the load-based molding, the current load value might not be representative of the resulting load when the TAO is actually executed. We choose to use the two molding policies hierarchically where the history-based molding was only considered if the load-based molding found that the load is high enough for any resource width. Although some tentative testing found this to be the most suitable option, an example of where this could be unsuitable is when the load-based molding chooses to resize to a resource width of two because the system load is low. This would be a bad decision if width two performs worse than width four for a specific TAO which the load-based policy would be unaware of as it does not consult the PTT. In addition to this, the molding is currently done after the scheduling extensions have affected the choice of core which could potentially affect performance. Both of these decisions could be explored further to find improvements to the task molding implementation.

7
Conclusion

The purpose of this thesis was to introduce heterogeneous resource-aware scheduling into an existing runtime called XiTAO. The goal of the runtime is interference-free resource-smart scheduling for a multicore environment. The runtime utilizes a notion of moldable tasks which allows for coarser scheduling of tasks onto a desired amount of resources through a given resource width.

In this thesis, we have presented different scheduling extensions to the XiTAO runtime to expand its homogeneous scheduling to consider a heterogeneous platform. Techniques from related heterogeneous scheduling methods together with moldable tasks from XiTAO were used in our suggested scheduling implementations. The extensions targets a single ISA heterogeneous multicore architecture and was evaluated on an ARM big.LITTLE architecture for performance increases.

A performance trace table was introduced to record execution times of tasks on the different cores with different widths. The first scheduling extension aimed to find the critical path of a task-DAG and schedule it on the high-performance big cores. This scheduling extension was implemented in two ways, one that finds the historically best performing cores from the performance trace table and one that requires information about the core structure of the platform prior to execution. The second scheduling extension used the performance trace table to find potential speedups from running on a high-performance big core compared to an energy-efficient LITTLE core before scheduling accordingly.

We also introduced a scheduling extension which targets the molding property of the runtimes tasks. This extension can be used together with the other extensions or in isolation. This molding extension considers both the load and the history of tasks to find an appropriate resource width.

The heterogeneous scheduling extensions was evaluated and compared to the XiTAO runtime scheduling without any extensions, the homogeneous scheduling. When an inappropriate resource hint is chosen, we see speedups of 28-182% depending on the parallelism of the DAG. This shows that the task molding extension is able to find an appropriate resource width efficiently. When an appropriate resource width is chosen, we observe speedups that range from 10% up to 45% from the schedulers that target criticality and the scheduler that targets the performance trade-offs. Overall, the difference in speedup mostly depends on the parallelism of the DAG but all of the scheduling extensions show an improvement over the original scheduler.

It would be interesting to cross reference our work on another platform in order to properly test the scheduling implementations and their portability. Evaluating on another platform would also allow us better access to different metrics and the scheduling could be evaluated in terms of energy-efficiency.

Although more research can be done to achieve even higher performance gains and potential energy-savings, the implemented scheduling extensions show that there is significant potential in considering the heterogeneity of the platform when scheduling. Furthermore, utilizing the notion of elastic places to schedule tasks with the most suitable amount of resources together with the correct resources is especially interesting when this is unknown prior to execution or when running applications with varying degrees of parallelism. Overall, the work presented in thesis shows good potential for resource-aware scheduling in heterogeneous platforms.

Bibliography

[1] ARM Limited. (2013) big.LITTLE Technology: The Future of Mobile (White paper).

[2] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, . . .* , vol. 13, no. 3, pp. 260–274, 2002. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=993206

[3] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures," *Proceedings of the 29th ACM on International Conference on Supercomputing - ICS '15*, pp. 329–338, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2751205.2751235

[4] M. Pericas, "Scalable and Locality-aware Resource Management with Task Assembly Objects," in *Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures (RESPA'15).*, 2015.

[5] M. Pericàs, "Elastic places: An adaptive resource manager for scalable and portable performance," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, pp. 19:1–19:26, May 2018. [Online]. Available: http://doi.acm.org/10.1145/3185458

[6] C. U. of Technology, "Low-energy toolset for heterogeneous computing (LEGaTO)," 2017. [Online]. Available: https://research.chalmers.se/en/project/7762 Last accessed: 2018-01-22.

[7] Maciej Drozdowski, *Scheduling for Parallel Processing.* London: Springer, London, 2009.

[8] A. S. Tanenbaum and B. Herbert, *Modern Operating Systems*, 4th ed., Mania J. Ho/ton, Ed. New Jersey: Pearson, 2015.

[9] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design.* Cornwall: Cambridge University Press, 2012.

[10] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, 1992.

[11] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "A simple load balancing scheme for task allocation in parallel machines," *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures - SPAA '91*, pp. 237–245, 1991. [Online]. Available: http://dl.acm.org/citation.cfm?id=113379.113401

[12] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, 1999.

[13] M. Pericas, "XiTAO," 2018. [Online]. Available: https://sites.google.com/site/mpericas/xitao Last accessed: 2018-01-23.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” *ACM SIGPLAN Notices*, 1998.

[15] Q. Chen and M. Guo, *Task Scheduling for Multi-core and Parallel Architectures*. Singapore: Springer Nature, 2017.

[16] H. Cheng, “A High Efficient Task Scheduling Algorithm Based on Heterogeneous Multi-Core Processor,” *2010 2nd International Workshop on Database Technology and Applications*, no. 3, pp. 1–4, 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5659041

[17] K. Chronaki, A. Rico, M. Casas, M. Moreto, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, “Task Scheduling Techniques for Asymmetric Multi-Core Systems,” *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 28, no. 7, pp. 2074–2087, 2017.

[18] D. Koufaty, D. Reddy, and S. Hahn, “Bias Scheduling in Heterogeneous Multi-core Architectures General Terms Algorithms, Performance,” in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 125–138.

[19] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, “Scheduling heterogeneous multi-cores through performance impact estimation (PIE),” in *Computer Architecture (ISCA), 2012 39th Annual International Symposium*, 2012, pp. 213–224.

[20] Linaro, “HiKey960 Development Board User Manual,” 2018. [Online]. Available: https://github.com/96boards/documentation/blob/master/consumer/hikey960/hardware-docs/hardware-user-manual.md Last accessed: 2018-05-04.

[21] Android, “SDK Platform,” 2018. [Online]. Available: https://developer.android.com/studio/releases/platform-tools.html Last accessed: 2018-02-13.

[22] “OpenEmbedded,” 2017. [Online]. Available: https://www.openembedded.org/ Url date: 2018-02-12.

[23] L. Foundation, “The Yocto Project, It’s not an embedded Linux Distribution, it creates a custom one for you.” 2018. [Online]. Available: https://www.yoctoproject.org/ Last accessed: 2018-04-12.

A
Linux Installation Guide

Please note that the following guide is based on the status of the Linux operating system for the HiKey960 board at the time of writing and the following information was gathered through forum threads. **This guide follows closely from that of the Base Firmware files and installation guide [1] and correlating forum threads.**

Preparation

- Items needed: Linux Host Machine, Hikey960 Board, Appropriate Power supply for board, Type-A to Type-C USB, UART to USB @1.5V.
- Potentially remove the modemmanager package in case this is installed as it might interfere with the installation process:

```
$ sudo dpkg -s modemmanager
$ sudo apt-get remove modemmanager
```

- Create a directory and download the images files for the UEFI. These can be found in the bootloader folder in the latest/wanted Linux build directory [2] or from the latest release version of UEFI [3] the UEFI should contain image files such as sec_xloader.img, l-loader.bin file and fip.bin file, a config file and a script called hikey_idt.
- Also download the corresponding boot and system(rootfs) images to the wanted Linux dist and place these in the folder. We used version 17 Debian image [4] together with the latest UEFI release(55). We do recommend trying to acquire a stable version at the time of installation as the Linux dist is being worked on.
- Download and install either ser2net or picocom to use as a console to the board through the UART:

```
$ sudo apt install picocom
```

Download the UEFI in Recovery Mode

- Change the switches to ON-ON-OFF(on switches 1-2-3) to place the board in recovery mode.
- Connect the UART-USB to the host machine, the HiKey960 boards have two UART connections where UART6 is used for serial console output. The

[1]https://github.com/96boards-hikey/tools-images-hikey960 Accessed:2018-02-07

[2]http://snapshots.linaro.org/reference-platform/embedded/morty/hikey960/100/rpb Accessed:2018-02-07

[3]http://builds.96boards.org/snapshots/reference-platform/components/uefi-staging/55/hikey960/ Accessed:2018-02-17

[4]http://snapshots.linaro.org/96boards/hikey/linaro/debian/17/ Accessed:2018-02-17

UART6 is a two wire UART connection on the pins 11 for TXD and 13 for RXD. Make sure the UART is visible through lsusb and/or dmesg|grep ttyUSBY. Y is some USB port of your host machine in our case this was ttyUSB1.

- Power up the board.
- Connect board to the Linux host using the USB-A to USB-C cable
- Check with commands such as lsusb and/or dmesg|grep ttyUSBX to see which device path the USB cable to the board is using, in our case this was ttyUSB0.
- Start your chosen console connection to the board, please make sure that you are using the correct device path as the UART connection for this:

```
$ picocom /dev/ttyUSBY -b 115200
```

- Navigate to the directory where the downloaded images.
- Chmod and run the script file to download and install the UEFI to the board. Make sure to use the correct device path as the USB connection for this:

```
$ chmod +x hikey_idt
$ sudo ./hikey_idt -c config -p /dev/ttyUSBX
```

- UEFI is now downloaded and flashed onto the UFS flash storage.
- At this time the board should switch to fastboot mode. Do not turn it off.

Flashing Linux in fastboot mode

- Located in the same directory as before fastboot the images for the UEFI installation:

```
$ sudo fastboot flash ptable prm_ptable.img
$ sudo fastboot flash xloader sec_xloader.img
$ sudo fastboot flash fastboot l-loader.bin
$ sudo fastboot flash fip fip.bin
```

- Navigate to the Linux images downloaded and fastboot these

```
$ sudo fastboot flash boot SOMEBOOTNAME.img
$ sudo fastboot flash system SOMESYSNAME.img
```

- Now the installation should be complete and ready to be used after reboot into normal mode(ON-OFF-OFF).
- *Note: Any errors occurring during the installation process will be displayed through your console UART connection so monitor this connection and use it for debugging if problems occur.*

Booting up Linux in Normal mode

- As the GRUB environment is installed the console will now give the possibility to enter fastboot mode to fastboot new images (f) or proceed to run the system image installed (automatic option if none other is pressed).
- Boot into the Linux dist and if password is asked for its root.
- You can now use your Linux dist.