Cooperative user- and system-level scheduling of task-centric parallel programs

GEORGIOS VARISTEAS

Licentiate Thesis in Information and Communication Technology
Royal institute of Technology, KTH
Stockholm, Sweden, 2013
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Licentiat examen 2013-09-27 i KTH Forum Kista.

© Georgios Varisteas,

Tryck: Universitetsservice US AB
Abstract

Emerging architecture designs include tens of processing cores on a single chip die; it is believed that the number of cores will reach the hundreds in not so many years from now. However, most common workloads cannot expose fluctuating parallelism, insufficient to utilize such systems. The combination of these issues suggests that large-scale systems will be either multiprogrammed or have their unneeded resources powered off. To achieve these features, workloads must be able to provide a metric on their parallelism which the system can use to dynamically adapt per-application resource allotments.

Adaptive resource management requires scheduling abstractions to be split into two cooperating layers. The system layer that is aware of the availability of resources and the application layer which can accurately and iteratively estimate the workload’s true resource requirements.

This thesis addresses these issues and provides a self-adapting work-stealing scheduling method that can achieve expected performance while conserving resources. This method is based on deterministic victim selection (DVS) that controls the concentration of the load among the worker threads. It allows to use the number of spawned but not yet processed tasks as a metric for the requirements. Because this metric measures work to be executed in the future instead of past behavior, DVS is versatile to handle very irregular workloads.
Acknowledgements

I would like to give special thanks to...

*Mats Brorsson*, Professor at KTH and my advisor
for giving me clear perspective

*Karl-Filip Faxèn*, Researcher at SICS and my co-advisor
for being patient and accessible and helpful with all sorts of programming troubles

*Christian Schulte*, Professor at KTH
for introducing me to a field of CS I came to love

...As well as everyone else who listened to my questions and spent the time to help me along the way of completing this project
# Contents

Contents v

List of Figures vi

## 1 Introduction and Motivation
  1.1 Scope ........................................... 1  
  1.2 Disposition of the thesis ................................. 3  
  1.3 Contributions ........................................... 3

## 2 Background
  2.1 Barrelfish operating system ............................... 5  
  2.2 Task-centric models and work-stealing scheduling ........... 6  
  2.3 Runtime schedulers ......................................... 7

## 3 Application runtime
  3.1 Scheduling task-based applications ..................... 9  
  3.2 Estimating a workload’s resource requirements .......... 12  
  3.3 Adapting to malleable resource allotments ............ 13

## 4 System-wide scheduling
  4.1 Effective system-wide load-balancing .................. 15  
  4.2 Process model ............................................ 16  
  4.3 Resource partitioning strategy ........................... 17

## 5 Summary of publications
  5.1 Publications .............................................. 19  
  5.2 Contributions ............................................... 19  
  5.3 Paper I ...................................................... 20  
  5.4 Paper II .................................................... 20  
  5.5 Paper III .................................................... 21  
  5.6 Paper IV ..................................................... 21

## 6 Related Work and Conclusions
  6.1 Related work ................................................ 23
List of Figures

3.1 Parallel execution starts at the center and spreads outwards. Each zone could spawn enough tasks to feed its outer one. .................................................. 10
3.2 Assuming an allotment of workers of certain size, the parallel execution starts at the center and spreads outwards. Each zone could spawn enough tasks to feed its outer one. The victim selection rules re-flow the tasks inwards via multiple alternative paths .................................................. 11
3.3 Illustrations of the execution of realistic parallel programs, top to bottom. Each rhombus is a parallel section. .................................................. 12
Chapter 1

Introduction and Motivation

1.1 Scope

The topic of this thesis is the effective scheduling of parallel applications on emerging multiprogrammed manycore architectures, combined with the use of a modern distributed operating system. Explaining this title from the end, the Barrelius operating system [5] has been investigated and used as the primary experimental platform. It is a novel approach to the old idea of the micro-kernel, being called the multi-kernel model [4]. The main concept is the deployment of a single micro-kernel per actual physical core in the system, with almost all services in the user space.

Multicore architectures are currently the norm; new architectures are pushing the boundary by employing one hundred cores or more on a single chip die. Such many-core designs create multiple issues with current scheduling techniques and operating system structure [40]. Namely, the inability of the operating system to allot the resources that can actually be utilized by a workload, resulting to over- or under-commit them [24, 42]. More hardware related issues make the move to these systems non-trivial. To name a few, inter-core communication latency and network-on-chip contention, memory locality and off-chip resource sharing. Two different features can assist in improving the utilization of large scale systems; multiprogramming or the ability to power off unneeded resources. Dynamically adaptive resource allocation is a promising solution that enables both functions [2, 9].

Systems with few cores and emerging manycore systems do not face the same issues when it comes to resource management and process scheduling. Parallel workloads can efficiently utilize small numbers of cores. Hence there used to be no need to restrict an application from requesting access to all available resources. Current operating systems, commonly time-share resources or migrate threads according to individual load balancing criteria. However, very few algorithms can be parallelized so as to persistently utilize tens or even hundreds of cores, making their static scheduling inefficient on large scale multiprogrammed systems [3].

Traditional resource management has kept the system and application scheduling sepa-
rated, leaving the former to blindly accept the resource requests an application makes [28].

However, the application’s runtime scheduler can be aware of the underlying workload’s actual resources requirements, while the system knows of the availability and status of such resources in the system [46]. Thus scheduling could be optimized if both levels cooperate [20]. Each of these two scheduling layers has with its own challenges. The application layer requires suitable programming models and runtime schedulers which can produce an accurate estimation of their actual resource requirements. The system must cost-effectively perform fair resource management for multiple applications.

Adaptive resource management requires the scheduler to map behavioral patterns to a variety of different workloads and resource configurations. The system must adapt to malleable application requirements and the application must be able to adapt to malleable resource allotments. Real life parallel workloads do not express a static level of parallelism; their ability to utilize a certain number of cores is fluctuating throughout their execution [29]. This complicates the development of generic abstractions that accurately estimate the resource requirements of any workload and alter the resource allotment accordingly.

On the system level, effective scheduling has to solve its own issues. First of all such a scheduler should be able to work with a variety of different runtime schedulers and be agnostic of how each distributes its workload. The problems attached concern identifying which exact resources to offer or remove from an adapting allotment. Moreover, global planning is not simple. It is usually modeled as a bin packing problem which is NP-hard. Many algorithms exist for solving or approximating a solution, either non-deterministic or heuristic. A way to avoid this situation is by scheduling each application independently. This means that single-application scheduling decisions should consider the overall system status but not affect active scheduling of the other running processes.

Scheduling based on requirement estimation feedback by the application’s runtime, can vary in effectiveness based on the accuracy of the estimation. Furthermore, an approximation of a workload’s status has an unpredictable duration of validity due to varying parallelism. Increasing the frequency of checking the status can minimize this unpredictability, simultaneously introducing excess overhead. Thus a final conclusion is that the scheduling decisions should not be absolute, leaving enough leeway to mitigate erroneous estimations by the applications.

This thesis presents a complete mechanism for the estimation of resource requirements and the dynamic adaptation of the allotment accordingly. The main principles governing the proposed design are as follows:

- Throughput: allotted cores should be maximally utilized or kept idle.

- Fairness: all simultaneously running apps should sacrifice no more than the same percentage of their expected performance.

- Efficiency: the amount of unnecessarily scheduling applications should be minimized.
1.2 Disposition of the thesis

The next chapter discusses earlier and related work. Chapter 3 discusses in detail the two scheduling layers this project deals with, identifying the issues and analyzing possible strategies. Chapter 5 is a summary of the attached publications. The thesis closes with chapter 6 iterating future goals and lessons learned.

1.3 Contributions

The contributions of this thesis are:

- A novel work-stealing scheduler using a provably efficient non-random policy.
- A fast, lightweight and accurate requirement estimation algorithm based on the non-random work-stealing policy.
- Implementation and evaluation of multiple mechanisms for the estimation of requirements.
- Scheduling abstractions for feedback based system-wide load-balancing and resource management.
Chapter 2

Background

2.1 Barrelish operating system

Barrelfish is a novel approach to a distributed OS [35]; it follows the multikernel model [4], having one independent microkernel per physical core. These multiple kernels form a distributed network. The system status is replicated across kernels using typical distributed algorithms although each kernel has its own set of resources and memory space. Inter-core communication is based solely on message passing hence uses no shared-memory; however, the use of shared-memory by applications is supported and a minimal POSIX-like API is provided.

It is important to clarify the user-space and kernel-space on Barrelfish. Contrary to typical OS architecture, in Barrelfish all services are run in user-space. Only each microkernel has access to the privileged mode, while it functions just as a device driver for the underlying processor. Thus, a third space has to be introduced, the application-layer, which includes end-user applications.

From the perspective of scheduling, Barrelfish provides very minimal functionality. There is no inter-core scheduling, which means no migration between kernels. Since there is one kernel per physical core, this means that threads are pinned to the core they are started on. Each microkernel employs a typical best-effort based scheduler switching between all applications it is hosting. Once an application is selected, a simple round-robin algorithm schedules one of its threads for execution.

This project focuses on the application layer, exploiting the message passing distributed operating system with a shared-memory based set of applications.

In Barrelfish spawned threads are statically positioned to their initial hosting core unless explicitly migrated by user-space code. That’s were our project lies as a unique intermediate layer. Although Barrelfish does not use shared-memory for cross-core system functionality, a userspace application is able to spread across multiple cores with all of its threads sharing the same address space. This model fits our design for pinned, one-per-core, shared worker threads.

Concluding, our design and implementation comes to complement the message-passing
platform of the Barrellfish operating system; it takes advantage of its intricate design, complementing the lack of needed scheduling functionality. Contrary to other popular OSs like Linux, Barrellfish matches very well the requirements of this project. Its independent and minimal components, enable the integration of extra functionality effortlessly, provide the foundation for in depth optimization and allow investigation of radically new structural designs.

2.2 Task-centric models and work-stealing scheduling

In recent years parallel programming has seen a sudden evolution [6]. Existing programming paradigms where enriched [12, 7, 31, 30] while many new were developed [22, 37, 36]. These can be separated into two main families, shared memory and message passing models.

Message passing models are based on the axiom that no two execution units should share memory. Parallelism is exposed via executing multiple cooperating processes, while all data are copied between them via messages. Data structures are kept synchronized through algorithms borrowed by the field of distributed systems. A good analogy of this model is a network of individual systems. Like with any such network message passing based parallel programs scale well, once the proper protocols for data exchange are in place. However, developing such protocols contains a certain overhead and complexity that is excessively large and can be disproportional to the complexity of the program itself.

Shared memory based programs are built using independent execution units called threads, that access the same memory directly. Data synchronization is constrained to the level of cache coherency, a function often provided by the hardware. The immediate benefit of such model is less complexity and easiness of development. However, such models are often victims to there own design; shared memory accesses require protection in order to avoid corruption or races. Such protection schemes need careful implementation so that the exposed parallelism is not decreased.

In comparing different ways of building parallel applications, a crucial criterion is how well their parallelism can be mapped and expressed. The most frequent obstacle is overcoming dependencies between blocks of code that could be executed in parallel, and therefore require data synchronization. Synchronization inherently serializes the execution as one thread has to wait for others to complete or catch up.

Shared memory based applications can be built using a variety of models. Most common has been the creation of a separate thread for each parallel work unit. However, the process of creating and destroying such threads can have a significant cost. Therefore for fine-grained workloads, where the parallelism can be in the thousands of simultaneous work units, such model would be inefficient. Alternatively, one could spawn a fixed pool of threads, called worker threads, that are successively provided with work.

Tasks-centric models [27, 32, 41] fall into the last category; they have been shown to express parallelism nicely and are capable to handle very fine grained loads. A task is an independent, self-contained and variably-grained block of code, programmed in the form of a closure; a function coupled with the appropriate arguments. Thus a set of tasks can
2.3. RUNTIME SCHEDULERS

and should be executed in parallel. Furthermore, they allow great flexibility in expressing
nested parallelism, since it is simple to create a new instance within an other. Alternative
models, like a parallel for loop, do not provide such functionality without explicit support
and complicated scheduling on behalf of the runtime. Usually iterations are split into
chunks; each chunk is executed sequentially by a different thread. With tasks, such a
chunk can be split into more, smaller ones recursively distributing the load across multiple
threads.

When working with a pool of worker threads, an important aspect is the algorithm
that handles the distribution of the parallel work units within them, usually referred to
as runtime scheduling. Two methodologies have been established. One is work-dealing,
where a single master thread is responsible for distributing and balancing the work among
worker threads. Such scheduling works well when the number of threads and the level of
parallelism are bound and considerably small. However there are issues of scalability once
either property rises beyond a manageable threshold.

The second method is work-stealing. It can be perceived as a decentralized version
of the former method, where its worker thread is responsible to acquire work units without
any centralized assistance or interference. The source of these units could be either a single
centralized queue of tasks or private queues per worker. It has been shown that the latter
option performs better, thus it has been adopted by most implementations. As the name
gives out, a worker (thief) steals work from other workers (victims) by peaking at their
private queues.

It now becomes obvious that the selection of victims can be critical in the successful
dissemination of the available work units. If done incorrectly, for example a subset of
workers only steal from within their closed group, it can lead to significant under-utilization
of available resources and loss of performance.

This project has focused from the start on task-based implementation of different work-
loads, executed using work-stealing scheduling. The subject of research has been to iden-
tify the points that affect the efficiency of the involved algorithms and proposed novel ideas
to optimize their efficacy.

2.3 Runtime schedulers

This thesis discusses features and capabilities added to work stealing runtime schedulers.
It does not investigate work-stealing in its entirety, rather specific aspects. For that reason,
the implementation has known schedulers, with established performance. This section
presents these schedulers.

WOOL

WOOL is a tasking library developed by Karl-Filip Faxén [18]. The programming model
involves defining functions and loops to be executed in parallel and includes three oper-
ations, call, spawn and sync. The first two call a task, the former sequentially like any
function, the latter will place it in the queue to be stolen or executed later. Sync, retrieves
the return value of the latest spawned task; if the synced task is stolen but its execution is not finished, the worker will *leapfrog*, try to steal a task from the thief of the task to-be-synced.

WOOL spawns and steals tasks at very low cost, resulting in very good overall performance; with fine-grained parallel workloads it can achieve linear speedup [19].

This runtime has been the foundation of the experimental implementations in this thesis.

**Cilk++**

Cilk++ is a tasking library developed by R.D. Blumofe et.al. [7], originating from the Cilk-5 runtime scheduler [21]. Its programming model is very similar to WOOL, although it employs a different scheduling strategy. Upon spawning a task, it is executed immediately by the worker. A continuation to the task that made the spawn is put in the task queue and can be stolen. To that extend it uses a cactus stack and requires a custom compiler, because the *continuation-passing-style* does not adhere to standards.

Recently Intel acquired Cilk++. They developed a new version called *Cilk Plus* which abandoned the cactus stack for standards compliance. Support for Cilk Plus has been integrated into the Intel C++ Compiler (ICC) with the 2010 iteration and into the GNU C Compiler (GCC) suite as of version 4.8.

Cilk++ and Cilk Plus have been investigated to apply the proposed algorithms in this thesis, but no implementation has been completed as of this writing. However, it is the foundation of other closely related projects.
Chapter 3

Application runtime

This section explores in more depth the application runtime scheduling abstractions presented. It includes the algorithms for the distribution of the tasks among available workers, the mechanism for estimating the requirements of a workload in terms of number of workers it can utilize and the adapting functionality when the size of that pool of workers changes.

Each step includes certain problematic aspects that affect efficiency; these are identified in the form of questions followed by an investigation of possible answers and the trade-offs therein.

3.1 Scheduling task-based applications

A task-based parallel application can be thought of as the combination of a set of tasks which are spawned either sequentially (flat parallelism) or recursively (nested parallelism) and a set of worker threads that can process those tasks. Scheduling of such applications can be characterized by two criteria: effective utilization of the available workers and minimization of the critical path.

The scheduling of a thread is effective when its execution progresses the overall execution of the workload. Consequently the host processing unit is effectively utilized. There are plenty of scenarios of ineffective scheduling of threads; they can be conceptually summed up in two categories. First when the amount of parallelism is not enough to employ all threads. Second, when the distribution of work among the threads is slow and non-uniform creating an unbalanced utilization of resources. The lack of parallelism can be measured relatively to the amount of allotted resources. It can be caused either by the commitment of too many resources or a change in the levels of parallelism in the workload itself. There are two methods for the distribution of work among worker threads, work-dealing and work-stealing. With the former, a single master thread distributes the load among all threads from a centralized queue of tasks. Work-stealing employs each thread with the ability to discover work; each worker has a local queue and work exchange is performed through stealing from remote queues. It has been shown that work-stealing sur-
passes the quality and performance of scheduling of work-dealing [8] and allows improved scalability to large scale [14].

The distributed nature of a work-stealing scheduler, makes the discovery of tasks non-deterministic. Moreover, such schedulers have been implemented using randomness for selecting candidate victims to steal from. At the start of any task-based application, a single worker spawns the first set of tasks. Random victim selection can lead to a delay in relocating tasks across all workers, leading to a non-uniform distribution and wasted cpu-time.

There are two popular optimizations to victim selection that improve performance and the steal success ratio. Sequential selection and Leapfrogging [44]. The latter selects a prior thief as a victim; it is employed during syncing, if the synced task is stolen but still executing. The intuition is that it’s more probable for a worker that stole a task to have spawned others that are stealable. With sequential selection, workers sequentially iterate the set of all available workers using a predefined order. Both these optimizations do not solve the delay in the initial stages of the execution, during which only one primary worker has spawned stealable tasks. For certain very fine grained and highly parallel workloads, the effect of this pattern is negligible. However, with coarser grained and irregular workloads, there can be significant loss of performance.

A workload is started on a single primary worker which spawns the first tasks. If all workers know this primary worker, they could select him as first victim and avoid initial delays. However, not very many parallel algorithms spawn enough tasks together to support a large number of workers. On the contrary, the tasks are spawned gradually. The single primary worker can feed a certain small number of other workers. Their overall capacity to spawn tasks is much larger allowing to feed even more workers. Figure 3.1 presents an extrapolation, where each stage is a zone. Of course such scheme only works when the parallelism in the workload is consistently enough and uniformly generated. Not many algorithms can be mapped to such model.

![Figure 3.1: Parallel execution starts at the center and spreads outwards. Each zone could spawn enough tasks to feed its outer one.](image-url)
3.1. SCHEDULING TASK-BASED APPLICATIONS

Figure 3.2: Assuming an allotment of workers of certain size, the parallel execution starts at the center and spreads outwards. Each zone could spawn enough tasks to feed its outer one. The victim selection rules re-flow the tasks inwards via multiple alternative paths.

A realistic representation of a common parallel workload, consists of an initial burst of parallelism followed by consecutive parallel sections of varying sizes as shown in figure 3.3 (a). Even this irregular load is only one pattern out of many. To accommodate such variety in a single scheduling abstraction there are two requirements, control of the concentration of tasks onto specific workers and a plan b. To control the concentration of tasks, the scheduler must control their relocation or in other words the pairs of workers that can or should steal work between each other. This can be achieved by non-random victim selection where the workers are guided by specific rules on where to steal from. Of course this implies positioning system among workers, much like the one in figure 3.1, where workers are directed to steal primarily from inwards. But as said before, this scheme does not work well with irregular loads. To achieve that, the rules must be augmented to allow secondary paths as in secondary victims at other directions.

A chain of steals can be envisioned like a flow of tasks being relocated from worker to worker. To accommodate any irregularity of the parallelism in a workload and achieve fast and uniform distribution across all workers, this flow must be self-regenerating. Assuming a load like in figure 3.3 (b), flowing the load outwards might leave inner workers without work fairly soon. To treat this, the load must be taken outwards and then re-taken back inwards to support the requirements of the inner nodes. Irregularity of parallelism can create a situation where a burst of tasks are spawned by a worker at any location in the allotment. Thus the victim selection rules must allow these tasks to be distributed across. Hence the victim selection ruleset must define alternative flow paths at any location.

In this thesis a deterministic victim selection algorithm (DVS) is presented that creates a flow like in figure 3.2. To achieve that scheme, all workers are classified based on their location relative to the primary worker that starts initiates the execution. Different classes of workers follow different victim selection rules and combined create the desired flow.
3.2 Estimating a workload’s resource requirements

In order to achieve adaptive resource management, the application’s runtime must be able to estimate its resource requirements and forward that information to the system. Acquiring such an estimation translates first in the definition of an appropriate metric. Related work has used cycles spent by the workers on useful actions [1, 9]. We have identified two inefficiencies with this approach. First the overhead of reading the cycle counter by each worker thread. Second is the use of past behavior to schedule the future. Work-stealing runtimes have reached a level of performing steal and spawn actions in just a few hundred cycles [19] making constant measuring of cycles a non-negligible overhead. Also the cycles spent can attest only to the utilization of resources up to that time point; if the parallelism is fluctuating, the assumption that past behavior will persist can be false.

Task parallelism is based on the idea that executing a task will result in spawning more tasks up to a certain recursion depth. Stealing a task relocates it to a new worker. The tasks spawned from that stolen task will be placed in the thief’s task-queue and can be stolen also. This process can be envisioned as a flow of tasks across the workers. A completely non-random victim selection can control this flow, providing concrete time bounds on the dissemination of the load. This type of determinism can help accurately
3.3 ADAPTING TO MALLEABLE RESOURCE ALLOTMENTS

control the concentration of the spawned tasks on specific workers thus predicting future resource requirements. The requirements metric in such scheme is the size of the task-queue of the workers (the number of stealable tasks, spawned but not processed or stolen). Acquiring such value does not involve overhead since its calculation is performed during the spawn and sync operations. Furthermore, the size of the task-queue reflects the amount of the work that has to be performed in the future. Hence increasing an allotment is a response to actual needs.

3.3 Adapting to malleable resource allotments

Adaptive resource management implies the need for the application’s runtime to dynamically adapt its scheduling to a fluctuating amount of resources. In the context of a work-stealing scheduler, these fluctuations take the form of added and removed workers. The action of adding workers is not problematic. They start fresh and steal work unaffected by any prior constraint. The removal of previously active workers though is not simple. "Which workers to remove?" is the first raised question. Additionally, there is the issue of removing workers with non empty task-queues. Those tasks must either be migrated to other task-queues, or the worker cannot be removed until they are processed.

Selecting candidate workers for removal should include those workers that contribute less to the progression of the execution. How to find these workers depends on multiple factors, like the requirements metric for quantifying contribution and inter-worker co-dependencies to preserve execution stability. For example the worker executing the critical path should not be removed as it is less probable to be insignificant.

Prior work has used cycles spent on useful actions as the requirements metric. Thus contribution can be mapped to the workers with the smallest such value. The assumption is that an arbitrary amount of resources could be removed by the system at any point. It has been shown that deploying more worker threads than available physical cores can degrade performance [42, 45]; thus the runtime must keep a record of the relative contribution of each worker to remove sufficiently enough. Such information involves significant overhead.

Deterministic victim selection allows to pre-define a specific subset of workers as preferred candidates, by keeping them utilized just enough. These workers steal less than they are stolen from, thus maintaining a shortage of tasks in their queues. Consequently the second issue of how to migrate non-empty task-queues is also answered.
Chapter 4

System-wide scheduling

Current and older operating systems have traditionally performed load balancing by allowing each workload to decide its own resource requirements. Legacy systems with a single or just a few processors resorted to resource time-sharing between processes or threads. Hence it was never required to restrict the amount of either.

Emerging many-core architectures have allowed for space-sharing resource management, where each workload receives an exclusive allotment of resources. However, most parallel workloads do not have static resource requirements; hence allotting fixed amounts of resources would lead either to loss of performance for the application or under-utilization of the resources. Being able to adapt the allotment is fast becoming a requirement.

The previous section explored the ability of an individual workloads to estimate their resource requirements, in terms of processing cores. It also discussed how the application’s runtime can adapt to a dynamically expanding or shrinking allotment. A significant part was the selection of which resources are best suited for removal, when such a necessity rises. This chapter explains the implications of selecting more resources to expand an allotment. A space-sharing strategy in a multiprogrammed system, poses constraints on the selection of resources to be allotted.

The concepts in this chapter regarding system-level scheduling and resource management have not been implemented and evaluated in this thesis. Nevertheless, these ideas have greatly influenced the design of the scheduling abstractions in the application layer and their implementation.

4.1 Effective system-wide load-balancing

In this section we focus on processing unit as resources and omit all others. Any system-wide resource manager must keep track of the resource allocations to individual processes. Managing the resources of a multiprogrammed system is a complicated but necessary task for systems that span enough resources to prefer a space sharing strategy. Two criteria have been identified for its performance characterization.
• **Fairness**: given limited resources, all simultaneously running applications should sacrifice no more than the same percentage of their *expected performance*.

• **Efficiency**: the amount of *underutilized* resources of applications should be minimized.

Expected performance is defined as the execution time of the same workload run in isolation with access to infinite resources. Multiprogramming will unavoidably result in processes being allotted less resources than they can utilize, degrading their performance.

The first criterion judges upon the ability of the resource manager to award each process with the appropriate amount, so that the percentage of performance degradation is the same for all. The second criterion assumes sufficient availability of resources and demands conservation in the allotments, allowing to power-off the unused resources.

### 4.2 Process model

At the system level, resource management decisions have to consider the status of all the physically available resources to avoid over- or under-committing them. Traditionally, the allocation of resources to a workload has been agnostic of the requirements of the workload. Either the system blindly follows the requests of the application, or it allots an amount delegating the responsibility of adapting to the application. However, an allocation of resources to a workload is the union of two sets: the set of physical resources the workload is allowed to utilize, plus the set of threads to run on them. Altering an allocation can mean modifying the size of either set. Current techniques have exclusively dealt with distributing physical resources among running processes. By prioritizing a space-sharing strategy, it is crucial that each thread runs on its own core exclusively. When resources are removed the runtime must optimally remove equally many threads. When resource are added, more threads must be created. Parallelism fluctuations happen at the millisecond scale, making either function cost-ineffective when repeated often. Scheduling that disables and re-enables threads is also not optimal.

For a workload whose parallelism has been mapped to tasks, these tasks are the only currency to be brokered. Ideally, there should be as many workers as there are tasks. In reality the runtime must use the resources that the system provides. Thus from the application’s perspective, resources should be threads. In some extent this is conceptually already in effect, although the responsibility of matching the thread count to the allotted resources is left up to the application itself. Considering a configuration where allotments are malleable at a rate of a few milliseconds, this model is not productive. The allotment of resources by the system should include the capacity to execute tasks without requiring the intervention of the application. Moreover, there is duality of roles in current parallel applications. Tasks represent an algorithm solving an arbitrary problem or performing a certain processing of data that can be independent of the execution platform. The runtime’s threads schedule the tasks for execution, with the effectiveness of the scheduling heavily dependent on the intrinsic aspects of the execution platform.
Effective system-wide resource management requires a new system software model. This model would replace the threads and process concept with a more lightweight execution abstraction of tasks. A process (or application) will then be represented by a collection of tasks and an allotment of cores (homogeneous or heterogeneous). The runtime executing the tasks would be offered by the system as part of the allotment; no scheduling logic should be included in the application itself. The size of the allotment should depend on the amount of parallelism it can currently use, on the load of the system and possibly on the priority of the application. The main advantages of this approach would be:

- Better resource utilization. No application can over-subscribe resources as only potential parallelism is expressed. An application only requests resources it can efficiently use.

- Compositional parallelism. Since the task abstraction expresses potential parallelism and is the only way of expressing concurrency, different parallel software components naturally can be composed to utilize even more parallelism.

### 4.3 Resource partitioning strategy

An important aspect of any system-wide scheduler is the method for distributing the physical resources between processes. There are two main strategies, namely time-sharing and space-sharing. The former allows sharing a resource between multiple processes via alternating time-slices. This has been the default strategy for most operating systems running on few or just one processor. The emerging abundance of processors and other resources allows for a different strategy, where processes are allotted resources with exclusive access.

Many parallel workloads have an upper bound on their achievable speedup. This implies that the smallest amount of resources that achieves that upper bound is optimal in terms of both performance and resource conservation. In the context of resource partitioning, only space sharing can create the connection between the optimal theoretical and actual allotment size. However, there can be competition of resources couples with very rapid very big fluctuations in parallelism. through these factors, it becomes clear that effective resource management is achievable only if both time-sharing and space-sharing strategies are employed.

An improved approach that fits the overall model described in this thesis, employs both strategies over different types of allotments. Processes are awarded resources optimistically; based on overall contention and the requirements estimation by each application’s runtime, the system provides slightly bigger allotments than it can or should respectively. For each application, a primary allotment is defined to include as many as there can be spared but no more than what was requested. The resources in this primary allotment are exclusively given. Then the system proceeds to expand this allotment with a small set of secondary resources that are time-shared, if such need arises. The intuition behind this secondary set is to absorb very rapid fluctuations and provide a foundation for the selection of resources to be added exclusively in the future.
Chapter 5

Summary of publications

5.1 Publications


Paper III: DVS: Deterministic Victim Selection for Improved Performance in Work-StealingSchedulers Submitted for publication.


5.2 Contributions

The contributions attributed to the work performed within the thesis include:

- A novel deterministic approach to work-stealing scheduling of task-based parallel applications, with mathematically proved properties and evaluated performance (Paper II).
- A novel method for the estimation of the resource requirements of task-based work-stealing parallel applications, with mathematically proved properties and evaluated performance (Paper II).
- Comparative evaluation of closely related work on semi-random work-stealing scheduling with and without a cycle based requirements estimating method (Paper III).
• Exploration of various trade-offs in achieving malleable allocation of resources (in terms of worker threads) (Paper III)

5.3 Paper I

Paper I presents a first experimentation with a distributed two level cooperative scheduler on top of the Barrelfish OS. A system-wide scheduler was implemented as a distributed network of multiple instances, each managing a unique subset of the available physical resources. These instances use message passing to share minimal state information and simple election algorithm for decision making on resource redistribution. The application layer is focused around a port of the WOOL work-stealing scheduler, augmented with an adaptation of ASTEAL, a workload requirements estimation algorithm developed by K. Agrawal [2]. The estimations are forwarded as feedback to the appropriate system scheduler instance. Over fixed intervals these instances perform global analysis and reallocation of resources as needed following a space sharing resource distribution strategy.

The scheme is evaluated by running pairs of specific benchmark applications that exhibit diverse and irregular fine-grained task-based workloads. All runs are performed with and without adaptation of the allotted resources. The results are positive to the space-sharing strategy, presenting better performance when resource allocation is adapted, minimizing unnecessary competition.

This work, although promising, can be thought of as a brute force application of the intuitive approach to such a two level scheduling scheme. It helped reach many of the lessons and design course discussed in this thesis. Follow up experimentation not in the paper but based on the orchestration it presents, revealed the scalability limitations of a global planning mechanism. Thus it helped identify the need for the application scheduler to be autonomous in deciding its requirements and the system level per-application asynchronously reactive to the adaptation requests. In overall it consists of the necessary step leading up to the ideas surrounding the rest of this thesis.

Individual contribution

For the purposes of this publication, I ported and adapted several technologies and benchmarking applications originally developed by their respective authors, to the Barrelfish OS. I decided on a testing strategy and configured an appropriate simulated evaluation environment for the Barrelfish OS. The results were analyzed and the paper written together with my advisor and co-advisor, both named as co-authors.

5.4 Paper II

Paper II consists of the sum of all lessons learned through the work up to and following Paper I. It reboots the project by presenting a mathematical model that redefined both the process model on the OS level and work-stealing as a solely deterministic algorithm, coupled with a novel method for the estimation of resource requirements of task-based
parallel application run with a decentralized scheduler (private per thread task-queues). For each component it presents the respective mathematical foundation and proves significant properties that theoretically support and verify their expected behavior.

The version attached was published as a technical report as a citable resource that helps alleviate the need of including the theory in follow up less theoretical work. Once the project is more mature, there are plans for a more extensive version to be submitted at a refereed venue.

**Individual contribution**

All work in this paper was performed by myself. The paper itself was written in cooperation with my advisor and co-author.

### 5.5 Paper III

Paper III expands upon the theoretical foundation of Paper II with an implementation of the deterministic selection policy. It describes and discusses the main concepts and benefits of DVS towards adaptive resource management. The main focus is a comparative evaluation of DVS. Two runtime work-stealing schedulers are implemented, WOOL-LF and WOOL-DVS. The first is the original WOOL scheduler that implements Leapfrogging; the second takes wOOL and replaces all victim selection parts with DVS.

Experiments are run on two platforms, a simulated one and real hardware. The simulated platform runs Barrelfish and models a 32 core mesh grid architecture. No memory hierarchy was added thus isolating the victim selection algorithm as the only implementation difference between the two runtimes. The second platform consists of a 48 core ccNUMA Opteron-based system, running Linux.

The presented results are promising. With fine-grained, highly parallel workloads WOOL-DVS achieves performance comparable to WOOL-LF on both platforms. With more irregular workloads the performance is improved, at an average of around 17% on both platforms. For a single allotment and workload on Barrelfish, WOOL-DVS performs worse than WOOL-LF; identifying points for further optimization.

**Individual contribution**

All work in this paper was performed by myself. The paper itself was written in cooperation with my advisor and co-author.

### 5.6 Paper IV

Paper IV expands upon the theoretical foundation of Paper II with an implementation of the resource estimation requirements algorithm (Ebb-3) coupled with a self-adapting implementation. It describes and discusses the main concepts and benefits of the proposed solution. The main focus is a comparative evaluation of Ebb-3 against related work (ASTEAL).
Three runtime work-stealing schedulers are implemented, namely WOOL, ASTEAL and Ebb-3. The first is the original non-adaptive WOOL scheduler that implements Leapfrogging; the second augments WOOL with the ASTEAL self-adapting mechanism; the third replaces all victim selection parts of WOOL with DVS and integrates the proposed self-adapting mechanism.

Experiments are run on two platforms, a simulated one and real hardware. The simulated platform runs Barrelfish and models a 32 core mesh grid architecture. No memory hierarchy was added thus isolating the victim selection algorithm as the only implementation difference between the two runtimes. The second platform consists of a 48 core ccNUMA Opteron-based system, running Linux.

The presented results are promising. With all workloads Ebb-3 achieves performance comparable to WOOL and ASTEAL on both platforms. It consistently provides more accurate requirements estimations, thus conserving resources. It manages up to 7% further reduction of wasteful scheduling in comparison to ASTEAL. With more irregular workloads Ebb-3 achieves on average 10% better performance compared to ASTEAL.

**Individual contribution**

All work in this paper was performed by myself. The paper itself was written in cooperation with my advisor and co-author.
Chapter 6

Related Work and Conclusions

6.1 Related work

Throughout this report there have been reference to work directly related to the ideas of this thesis. Nevertheless, there are plenty other projects investigating ideas that are orthogonal, complementary or complete replacements to the contributions of this work. This section iterates and discusses them.

Task-centric models and work-stealing schedulers have been the focus of much research the past few years. Tasks are logical entities, not bound to hardware, which provide a large level of flexibility in expressing parallelism. Also, the distributed nature of work-stealing and its documented high performance have led to apply it in a series of diverse environments [13, 14, 16]. Many have successfully experimented with alternative methods for more intelligently controlled selection of victims.

CAB (Cache Aware Bi-tier Task-Stealing) [11] focuses on avoiding cache pollution resulting from the random victim selection. Through the execution directed acyclic graph (DAG) it calculates data dependencies between tasks and schedules them so that they share the same caches. CATS [10], a follow up to CAB, provides the same bounds but great performance benefits and can handle more complicated DAGs. SLAW (Scalable Locality-Aware Work-stealing) [23] provides and uses locality awareness in a scheduler that can choose to follow either work-first or help-first on a per-task basis. Semi-random victim selection is being used, with the search space being restricted to a specific subset for locality preservation. HotSLAW [33] expands on SLAW, and introduces HVS (Hierarchical Victim Selection) increasing the locality benefits. Different components create a hierarchical ladder. Workers arbitrarily pick victims on the bottom of the ladder (highest locality) and move upwards until a steal is successful. Quintin et. al. [38] also embraces the idea of hierarchy, presenting two variants of a work-stealing scheduler. PWS restricts the victim search space to immediate neighbors, based on the hardware interconnect network. HWS applies a more complex notion of hierarchy in clusters where its is possible to transfer chunks of tasks between clusters while taking advantage of network properties to maximize performance. Saraswat et al. [39] has also tackled the problem of deploying a
work-stealer on distributed memory and at great scale of cores, through the introduction of lifeline load balancing. Lifeline graphs denote a connectivity network between workers as nodes. After a certain threshold of failed steals a node becomes inactive while informing its lifeline. Inactive nodes are not picked as victims until their lifeline provides work and reactivate them.

Several projects have targeted the topic of adaptive resource management for multiprogrammed systems. Adaptive Thread Management [25] optimizes the number of resources to data-parallel loops in automatically parallelized programs [26] during execution, according to the online comparison of current speedup against expect values. Adaptive Multi-block PARTI runtime library [17] follows a similar approach, providing great portability. Dynamic feedback [15] provides a compiler assisted optimization scheme, adapting to different compiler generated synchronization policy on alternating phases. ADAPT (Automated De-coupled Adaptive Program Transformation) [43] applies loop optimizations via dynamic recompilation, it is based on the Polaris parallelizing compiler [34].

Yangjie Cao et. al. in [9] work towards the same goal of adapting resources to workload requirements. They employ a mechanism that is efficient in estimating the aggregate resource utilization of all threads in a single application. They however do not sufficiently consider the question of which resources should be added or removed.

6.2 Conclusions

This thesis addresses two major components for the effective resource management of task-based parallel applications on multiprogrammed many-core architectures. The first component is a mechanism for the low-overhead and accurate estimation of resource requirements by a work-stealing runtime scheduler. The second component is a novel approach for the effective dynamically adaptive resource management of such systems, as fair among all running processes and efficient in conserving resources without necessarily sacrificing performance.

Future work is separated into two parts. First the implementation and extensive evaluation of the complete system with multiprogrammed test configurations. Finally the augmentation of the resource estimation metrics to account for the individual task granularity. Task granularity cannot be speculated offline or online; it requires information from both stages which can be achieved via dynamic recompilation. The required information are a taxonomy of the available tasks via offline profiling combined with data that can affect the parallelism of the task (i.e. the number of iterations).
Bibliography


[29] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casca


