# Dynamic Inter-core Scheduling in Barrelfish: avoiding contention with malleable process domains

Georgios Varisteas
KTH Royal Institute of Technology
yorgos(@)kth.se

Mats Brorsson
KTH Royal Institute of Technology
matsbror(@)kth.se

Karl-Filip Faxèn
Swedish Institute of Computer Science
kff(@)sics.se

## ABSTRACT

Trying to attack the problem of resource contention, created by multiple parallel applications running simultaneously, we propose a space-sharing, two-level, adaptive scheduler for the Barrelfish operating system.

The first level is system-wide, existing inside the OS, and has knowledge of the available resources, while the second level is aware of the parallelism in the application. Feedback on efficiency from the second-level to the first-level, allows the latter to adaptively modify the allotment of cores (domain) thus intelligently minimizing time-sharing.

In order to avoid excess inter-core communication, the first-level scheduler is designed as a distributed service, taking advantage of the message-passing nature of Barrelfish. The processor topology is partitioned so that each instance of the scheduler handles an appropriately sized subset of cores.

Malleability is achieved by suspending worker-threads. Two different methodologies are introduced and explained, each ideal for different situations.

## 1. INTRODUCTION

Most research on parallel programing models has focused on running parallel applications in isolation. Although helpful in investigating the fundamental properties of parallelization and multi-core architectures, this approach neglects the major issue of contention. In real-life situations a machine typically executes multiple parallel processes, which have to fight for the same system resources.

Barrelfish is a novel approach to the idea of **multi-kernel** operating systems [**?**]. It only uses message passing for inter-core communication and is not dependent on any cache coherence mechanism. Thus it guarantees scalability and portability even on heterogeneous systems. For scheduling, the RBED algorithm is employed[**?**] for ordering intra-core thread execution. Each process is allowed to create threads arbitrarily on a specific set of cores which is called its **domain** and it is mutable. However, there is no inter-core scheduling or, in other words, any intelligent control over the distribution of these threads onto the cores of a program's domain. So, very frequently the combined load becomes highly unbalanced, without system-wide knowledge and control, eventually under-utilizing the system. An obvious example, as depicted in figure 1, is when two programs of differing amounts of parallelism each asks to use the whole system. Also, it still remains a fact that most real-life applications can not be parallelized to the extend that they can efficiently utilize a many-core or even a multi-core system. In most programs the amount of parallelism that exists is fluctuating throughout their execution.

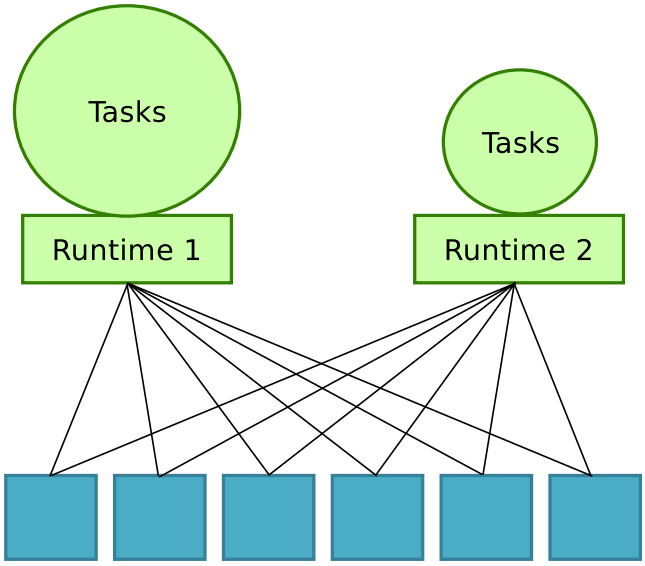


**Figure 1: Highly parallel runtime 1 and not so parallel runtime 2, are allowed to share the same amount of resources in the absence of any inter-core, system-wide scheduling.**

Nevertheless, message-passing although a scalable programming paradigm, it comes with a significant overhead in design and implementation. Shared memory based programming models, and more specifically task-based models, are easier to use for implementing the most complicated programs. It is a basic assumption of this project that both message-passing foundation of the operating system and various shared-memory-based programming models for building applications can be combined in a way that exploits the best attributes of both.

To that effect, we propose a **two-level adaptive inter-**

**core scheduler**. As described by Feitelson [**?**] schedulers for parallel systems can be designed in two levels in order to decouple resource allocation and resource usage. It is also simple to use this duality of the scheduler to combine message-passing (at the system-level) and shared-memory (at the application layer). As shown in figure 2 we add a **system-level process scheduler** as a completely new scheduling layer. It allots processors to processes (called "jobs" in [**?**]).The second level is a custom **user-level thread scheduler** which distributes a process' threads onto its allotted processors. The **adaptive** keyword is used to describe the malleability of the scheduling; the runtime provides feedback to the system-scheduler, allowing for dynamic [**?**] change of the allotted processors during execution.

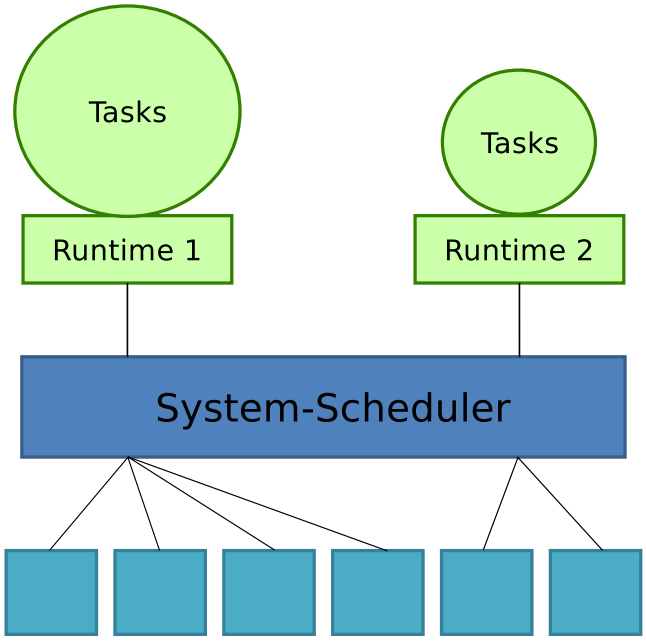


**Figure 2: The new system scheduler recognizes the lack of parallelism in runtime 2 and distributes the resources accordingly.**

# 2. TWO-LEVEL-ADAPTIVE SCHEDULER

For the creation of threads and the distribution of work the user-level scheduler employs the work-stealing task-based programming paradigm [**?**]. Work-stealing is an effective way to to implement a thread scheduler. Threads, called workers, execute fine-grained tasks while possibly spawning new ones. When necessary they independently acquire work by *stealing* available[1]tasks from other workers selected at random. However there are two specific situations where such models can lead into inefficient use of various system resources.

First of all, the efficiency of most work-stealing algorithms is fluctuating by definition. Resources are wasted when workers are trying to acquire work while none is available (**wasted-cycles**). The number of algorithms which expose a constant amount of parallelism throughout their execution and thus can utilize a fixed amount of workers, is quite small.

[1]All spawned tasks are by default marked as *stealable* until the worker that spawned them initiates processing

Also, it is an open question whether it is better to have multiple processes time-sharing system resources, or accomplish space-sharing by reducing the number of active workers.

Finally, it is very frequent for different threads of the same process to communicate with each other. Take for example the procedure of work stealing. If the worker threads are not scheduled for execution simultaneously on different cores, then inter-core communication (for stealing work) will require context switching; this produces a significant overhead. In the worst case scenario there can be a deadlock if the developer is not careful with the specifics of the underlying architecture.

## 2.1 Overall design

Figure 3 presents a snapshot of an execution of the system, indirectly visualizing the proposed design of the two-level scheduler. Each column represents one core. Going from the bottom up, the CPU driver is the hardware-software interface, tailored to the specific architecture of the underlying processor. the ability to have a different CPU driver for each kernel allows for barrelfish to be seamlessly deployed on heterogeneous architectures; a very useful out-of-the-box feature. The monitor is the boundary between kernel and user space, responsible for executing all processes; also, all intra-core communication is handled by the monitor. Among all other services, the system-level scheduler exists right on top of the monitor. For simplicity the figure presents one instance per core although this is not true as explained in section 2.5. In the figure there are two processes running; the scheduler has space-shared the system as much as possible but allowed core 1 to be time-shared.

## 2.2 Existing scheduling in Barrelfish

The existing RBED thread scheduler in barrelfish performs a decent job in deciding the execution order of the threads for each processor. It is combined with the proposed inter-core scheduler for handling time-sharing situations. Although the goal of this project is to space-share the system, situations where that is not possible can occur very easily. Section 3 presents such situation where time-sharing is actually used in order to provide the malleability of tasks.

## 2.3 Adaptive work stealing

Each process starts with an initial desire of $d_i$ processors. The *thread-scheduler* counts the *steal-cycles* (searching for work) and *mug-cycles* (stealing work); the sum of those is the amount of **wasted cycles**. The thread scheduler calculates the sum of the wasted-cycles of all of its workers, while also keeping track of the most inefficient worker. At the end of each quanta the *system scheduler* receives this metric tuple from all programs $p_i$ and decides on an allotment $a_i$ for each. The decision is based in comparing the current *wasted-cycles* to total processor cycles, characterizing the thread as **inefficient** or **efficient**. In parallel, if $a_i < d_i$ the process is **deprived**, otherwise **satisfied**.

The decision policy [**?**] is as follows:

- **Inefficient**: overestimation. The allotment is decreased for the next quanta.

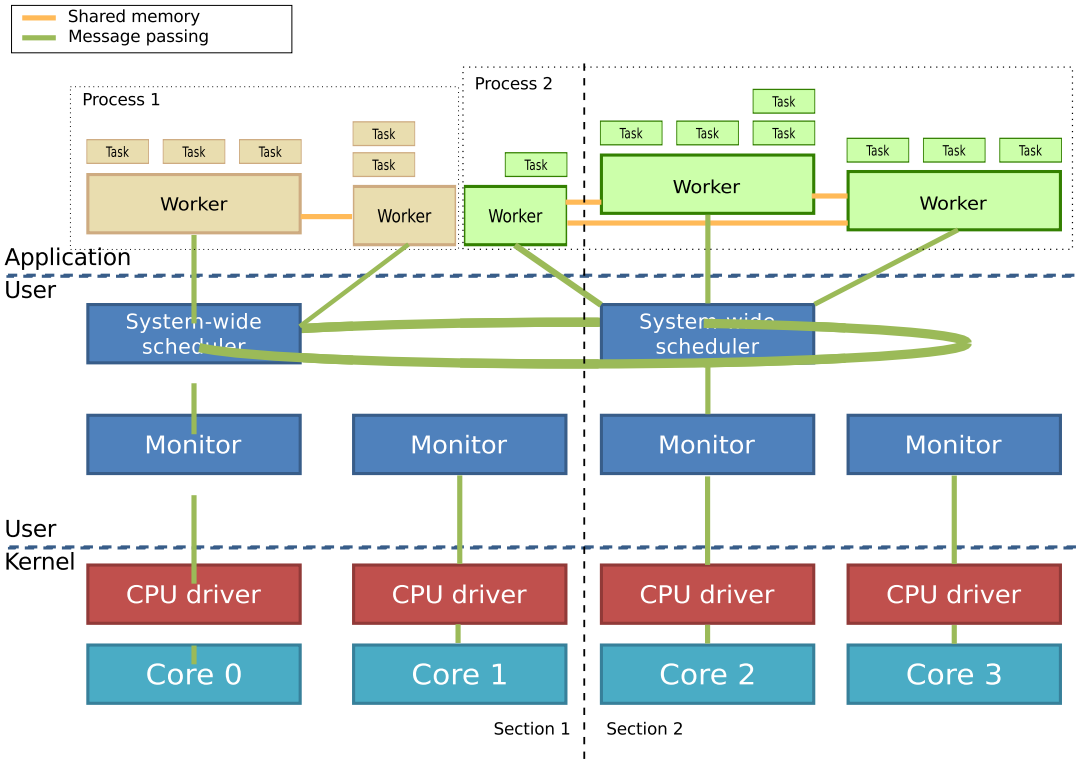


**Figure 3: A snapshot of the system, with two processes time-sharing the same core.**

- **Efficient and satisfied**: underestimation. The allotment is increased for the next quanta.
- **Efficient and deprived**: balanced. The allotment remains the same for the next quanta.

When a program is characterized as *efficient and satisfied* it means that there was excess amount of tasks in comparison to the available worker threads. This means that the program could effectively use more worker-threads. In contrast a *efficient and deprived* program although having enough work for its worker-threads, it does not indicate that more workers could be effectively used.

## 2.4 User-level thread scheduler

At the current stage we are using customized versions of two well-established task-based programming models. Both utilize work-stealing algorithms for work distribution. The first such model is *WOOL* [**?**], which is extremely fast and efficient in system-resource utilization. Next, *Cilk++* [**?**] is also investigated; it provides sufficiently fast applications but it can accommodate for the malleability property we are introducing. These two programming models although looking very similar in syntax and functionality, they have significant differences on how each spawns and syncs tasks. These are introduced in more details in section 3.

Apart from porting them to the Barrelfish operating system, the customizations done are focused in gathering and forwarding the required efficiency statistics to the system-level scheduler. The thread scheduler, by employing adaptive work stealing as introduced in [**?**], it provides feedback about the job's parallelism and efficiency to a process scheduler at regular intervals called **scheduling quanta**. The main task of this scheduler is to perform **work-stealing** while keeping track of efficiency. This metric is forwarded to the system scheduler at the end of each quanta.

## 2.5 System-level process scheduler

The *system scheduler* runs on top of the monitor which allows it to have system-wide knowledge of available resources. It collects the efficiency statistics from all running processes and repartitions the available processors among them. Such a mechanism would however act as a bottleneck for the system.

This is why the system scheduler is implemented as a distributed service. The processor topology is partitioned into sets of cores called **sections**. One instance of the system scheduler is overlooking each section, while exchanging necessary information with all other instances; a sample execution can be viewed in figure 4. A system scheduler instance is called **Section-Instance** or simply **SI**.

All SIs share the knowledge of two tables. First is the **ownership table** which holds the partitioning, meaning which cores are owned by which section. Assuming that the number of cores is fixed, this table is immutable. The other table shows which sections own cores for which processes. This is a simple one-to-many table, mapping processes to sections; it is called the **participation table**.

The basic principles of this service are:

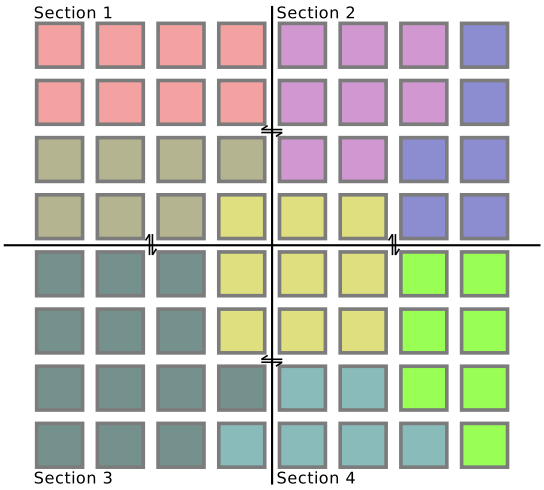


**Figure 4: A fully utilized processor divided into four sections. The process running on the central cores has spread across all sections; section 4 has primary control over it due to owning the majority of its cores.**

1. Each section is given a priority number that doubles as its unique ID.
2. A section should consist of at least two cores, except if only one is present.
3. Sections are not necessarily of equal size.
4. There is only one SI per section.
5. Processes can be allotted cores from multiple sections. Leader election decides primary SI:
   - A SI has primary control of a process, if it owns the majority of the cores it uses.
   - Otherwise, leader is the SI with highest priority number.
6. Thread schedulers communicate only with the SI that owns the core that hosts the process.
7. SIs can request information from other SIs, regarding processes or even specific threads.
8. SIs voluntarily share only the absolute necessary amount of information.
9. The only global knowledge is the *participation table* and the immutable *ownership table*.

Whenever the allotment of cores for the process is changed and if its entry in the participation table changes, a limited version of the leader election is executed. It is limited because the leader can be mostly predicted. Since each SI has a copy of the *ownership table*, it knows if the change can affect its reign and if yes who the new leader is. The priority of each core is also its Id in the aforementioned table. Thus exchanging new information is only necessary if the leader changes.

Over fixed intervals each SI decides on the program allotment. For the programs that have been characterized as *inefficient* their most inefficient core is considered as *available*. For the programs that have been characterized as *efficient and satisfied* it will try to allot one extra core. This will "ideally" be a *available* core of a program that has been characterized as *inefficient*. First it will check the programs it is primary of while broadcasting to the other SIs. If no other program is characterized as *inefficient* then it will select the most inefficient core of a *efficient and deprived* program to be time-shared between them. The actual algorithm is presented in figure 5.

**allot**(SI)
1: **for all** $p_{es} \in$ efficient and satisfied **do**
2: $r \leftarrow broadcastForAvailableCore(SI)$
3: **if** $r == NULL$ **then**
4: $r \leftarrow getCoreToTimeshareWith(SI)$
5: **end if**
6: $cores(p_{es}) \leftarrow r$
7: **end for**

**broadcastForAvailableCore**(SI)
1: **if** $inefficient \in myPrograms(SI)$ **then**
2: $r \leftarrow getAvailableCore(SI)$
3: **else**
4: **for all** $SI_i \in \{sections() \setminus SI\}$ **do**
5: $r \leftarrow getAvailableCore(SI_i)$
6: **end for**
7: **end if**
8: **return** $r$

**getAvailableCore**(SI)
1: **for all** $p_{ie} \in$ inefficient **do**
2: **if** $count(cores(p_{ie})) > 1$ **then**
3: **return** $mostInefficientCore(p_{ie})$
4: **end if**
5: **end for**

**getCoreToTimeshareWith**(SI)
1: $p_{ed} \leftarrow$the "Efficient and Deprived" program currently allotted the most cores
2: **return** $mostInefficientCore(p_{ed})$

**Figure 5:** pseudocode of the algorithm used to decide the allotment of cores for each program.

## 3. MALLEABLE PROCESS DOMAINS

Altering the initial allotment of processors translates into dynamically **creating and *removing* workers**. Since creating threads can be a very costly process, instead of destroying the worker-thread, it can be put to sleep; such workers will be called **suspended**. Although seemingly simple, it encapsulates the dangers of throwing out completed work but also deadlocking the rest of the workers. One obvious way to treat this is by not allowing to trim a processor from a process' allotment before the underlying worker has synced its current execution tree. In such a scenario there is no need for any bookkeeping. However, when the task has spawned

numerous other tasks it is very inefficient to have to wait for the whole tree to sync back. Thus it is mandatory to allow suspending workers at any given point. My preliminary research has revealed two distinct ways to accomplish this.

The first methodology revolves around task-based models like *WOOL*, which use the stack and busy waiting. The use of stacks makes task migration quite difficult and costly, as it involves a lot of unwanted bookkeeping across all workers that have stolen work from a *to-be-suspended-worker*. What could be done in this scenario is to **migrate the worker thread** onto a different core, preferably one hosting the *most inefficient worker thread*. The worker then can be suspended after its whole execution tree has been synced. This process shall be called **lazy-suspension**.

In contrast, programming models like *Cilk++*, utilize **continuation passing style (CPS)** instead of stacks, which allows for actual task migration. All of the application state is stored in the heap (shared memory) so any worker can take over the processing of a task; it is merely a matter of exchanging specific pointers between the workers. In this scenario the worker thread can be immediately suspended. This method shall be called **immediate-suspension**.

There is no better approach to the problem of malleable domains. Both methods have their trade-offs. **Lazy suspension** is easy to implement and maintain but is not the most effective. **Immediate suspension** is much harder to implement and requires a higher synchronization complexity, while being more effective.

# 4. PHASE-LOCK GANG SCHEDULING

One important scheduling paradigm introduced with Barrelfish is Phase-Lock Gang scheduling [**?**]. It involves an efficient gang-scheduling algorithm, tailored to the unique nature of Barrelfish as a distributed multi-kernel operating system.

> Core-local clocks are synchronized out-of-band and schedules coordinated, so that kernels locally dispatch tasks at deterministic times to ensure that gangs are co-scheduled without the need for expensive inter-core communication on every dispatch; this feature is being implemented as an extension to the RBED scheduler.

Such a mechanism can accommodate for the various situations where there cannot be **ideal partitioning** of the system, having only one worker-thread per core in respect to the whole system. Such situations are plenty, having less physical cores than running processes or having multiple highly parallel programs which can fully utilize the system, are among the most obvious

# 5. EVALUATION DISCUSSION

This project is ongoing research with very few preliminary results until of this writing. As it is not self contained, its completion is dependent on specific aspects of the Barrelfish operating system which are still in progress. Moreover, this research is meant to target many-core systems like Tile64Pro; our effort to port Barrelfish to this architecture is underway but not yet complete. In overall we are using a variety of architectures, from 8 core Intel i7 to 4x12 core AMD opteron to Tile64Pro.

Evaluation is usually a product of comparison between the established and the new approach; due to the current absence of any system-wide inter-core scheduling in Barrelfish, finding comparison points are not that simple. On one hand this project could be viewed as a promising first step. On the other we are planning to extend to outside the Barrelfish spectrum, thus indirectly evaluating the OS itself as a worthy research platform for multi-core systems

Thus the main evaluation points are to first prove the obvious ineffectiveness of having no load-balancing mechanism across a many-core system by running a diverse collection of programs with and without our scheduler. The diversity is mainly focus on the levels of overall parallelism in the programs but also on whether it is constant or fluctuating.

The Linux kernel is very aggressive in load balancing the system by frequently migrating threads. It is a process that Linux does very efficiently. Hence, it is a very powerful opponent which we have selected as the main comparison point against the final version of our design and it is going to be the focus of our evaluation.

# 6. CONCLUSIONS

In high-load many-core systems, it is worth comparing the efficiency of having threads of different processes time-share a core in contrast to space-sharing by fluctuating the number of worker-threads. This project combines both scheduling implementations and because barrelfish provides a complete, custom tailored environment, this project can eventually allow insight at all levels; from the operating system all the way up to the application layer.

Moreover, the existence of two different methods for thread suspension (lazy and immediate) in relation to Barrelfish, can handle cooperatively a variety of different situations. Additionally, it different types of programs benefit the most from different scheduling methods, thus this complexity if intelligently applied can lead into various insights.

The current status of this projects finds *WOOL* ported to Barrelfish. Unfortunately time wasn't enough to have any meaningful evaluation especially since there is no appropriate point of comparison. The ported version of *WOOL* requires little modification to be integrated with Barrelfish and the described schema as the user-level scheduler. Implementation of the system-level scheduler is underway, while experimenting with various practices for its distributed aspects.

# 7. FUTURE WORK

Our immediate goals, focus on evaluating our initial hypothesis of contention in barrelfish under high load, without any system-wide scheduling by using WOOL-based testcase programs and advanced simulation environments.

Moreover, this idea has been based on some assumptions that restrict the application spectrum. Generalizing can be

great optimizations and extensions. Three important examples are:

- Making the schedulers locality-aware while work-stealing (user-level) and allotting cores (system-level).
- Handling the absence of shared-memory support by the underlying architecture.
- Make use of processor capabilities as criteria on the decision of core allotment for the deployment on heterogeneous architectures.

Finally, a future goal is to allow for the aforementioned distinct suspension types (and their underlying programming models) to cooperatively work together as one system that intelligently adapts to various different needs.

# 8. REFERENCES


[1] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26(3):1–32, September 2008.

[2] Andrew Baumann, Paul Barham, P.E. Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, A. Sch\\"upbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 29–44. ACM, 2009.

[3] Andrew Baumann, Rebecca Isaacs, and Tim Harris. Design Principles for End-to-End Multicore Schedulers Context : Barrelfish Multikernel operating system. *Group*.

[4] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *ACM SigPlan Notices*, volume 30, pages 207–216. ACM, 1995.

[5] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Computing*, pages 1–29, 1994.

[6] S.a. Brandt, S. Banachowski, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. *Proceedings. 2003 International Symposium on System-on-Chip (IEEE Cat. No.03EX748)*, pages 396–407.

[7] S.H. Chiang and M. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *Job Scheduling Strategies for Parallel Processing*, pages 200–223. Springer, 1996.

[8] K.F. Faxén. Wool-a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2009.

[9] DG Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). *IBM Research Report RC19790 (87657) 2nd Revision*, 16(1):104–113, May 1997.