Resource management for task-based parallel programs over a multi-kernel

BIAS: Barrelfish Inter-core Adaptive Scheduling.

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, running close to the OS' kernel, and has knowledge of the available resources, while the second level, integrated into the application's runtime, is aware of its type and amount of parallelism. Feedback on efficiency from the second-level to the first-level, allows the latter to adaptively modify the allotment of cores (domain), intelligently promoting space-sharing of resources while still allowing time-sharing when needed.

In order to avoid excess inter-core communication, the system-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 suitable for distinct programming models and applications.

Preliminary results are quite promising and show minimal added overhead. In specific multiprogrammed configurations, initial experiments proved significant performance improvement by avoiding contention.

1. Introduction

Most research on parallel programming 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, multiple parallel processes are running simultaneously which have to fight for the same system resources.

Barrelfish is a novel approach to the old idea of a distributed operating system. It follows the **multi-kernel** model [3], according to which there is one micro-kernel per each physical core in the system. These kernels work as one distributed operating system. All services are shared. This design provides two great features right away; **portability**, since it can be deployed on any architecture as long as each available processor is supported and **scalability**. The latter is an outcome of the OS' most basic characteristic, it assumes no shared memory thus employing only message passing for inter-core communication. Hence there is no dependence on complicated cache coherence mechanisms.

Thread scheduling in Barrelfish in currently not very evolved; each kernel uses the RBED algorithm[6] to order the execution of the threads that are time-sharing its underlying core. However there is no automated mechanism for migrating threads between kernels. Although each process is allowed to create threads arbitrarily on a specific set of cores, which is called the **domain** and is changeable, there is no inter-core scheduling done by the system, 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, occasionally under-utilizing the system. An obvious example, as depicted in figure 1, is when two programs, of different amounts of parallelism each, request usage of the whole system. Furthermore, 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.

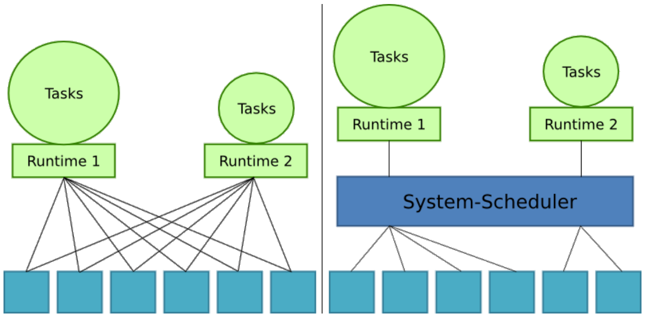

Figure 1. Highly parallel runtime 1 and not so parallel runtime 2, are allowed to share the same amount of resources in the absence of any inter-core, system-wide scheduling. We propose a new scheduling layer which recognizes the lack of parallelism in runtime 2 and distributes the resources accordingly.

Nevertheless, message-passing although a scalable programming paradigm, brings 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

RESoLVE '12 March 3 2012, London, UK.

most complicated programs. It is a fundamental assumption of this project that both the message-passing foundation provided by 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 this end, we propose a **two-level adaptive inter-core scheduler** (**B.A.I.S.**). As described by Feitelson [9] schedulers for parallel systems can be designed in two levels in order to decouple resource allocation and resource usage. This design simplifies mapping the aforementioned duality into the implementation; a scheduler that combines message-passing at the system-level and shared-memory in the application layer. As shown in figure 1 we add a **system-level scheduler** as a completely new scheduling layer. By knowing the system-wide availability of resources it allots processors to processes (called "jobs" in [2]). The **adaptive** keyword is used to describe the malleability of the scheduling. Given a significant execution time length, the domain of each application can and will be modified dynamically [7] during its execution, according to the overall requirements of the system. The second level consists mainly of the application's runtime, augmented to provide feedback on *efficiency* and *resource requirements* (**desire**) to the system-scheduler. The desire of the application is based on its awareness of the parallelism that the exists in the application

The runtime systems we are using are well established task-based parallel programming models, mainly *WOOL*[8] but also investigating *Cilk++*[5] and *OpenMP*. These are responsible for distributing a process' threads onto its allotted processors.

Preliminary evaluation and results have been positive. At the time of this writing we have tested a first naive and unoptimized implementation which however has a very small overhead in cycles. Also, first trials of contented configurations showed that our scheduler allowed applications to utilize resources better and increase performance.

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 [4]. Work-stealing is an effective way 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.

The efficiency of most work-stealing algorithms is theoretically not steady. 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. 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 delays. 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 2 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[2]; a very useful out-of-the-box feature. The monitor is the boundary between kernel and user space, responsible for executing all processes; also, it handles all inter-core communication and it facilitates inter-service interaction. In summary it is the monitor that transforms the set of independent kernels into one distributed operating system. Among all other services, the system-level scheduler exists right on top of the monitor. In the example of figure 2 there is one instance per two cores. In the application layer there are two processes running. The small squares depict tasks spawned by each application. Each is bound to its *local* system scheduler instance, with every worker being able to exchange information with it if needed. The scheduler has space-shared the system as much as possible but allowed core 1 to be time-shared. This is because in this hypothetical scenario application two has excess amount of tasks that can utilize 3 cores while application 1 is also efficient enough not to be deprived of its second core. The decision policies are described in more detail in section 2.5.

2.2 Existing scheduling in Barrelfish

The existing RBED thread scheduler in Barrelfish performs an adequate 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 configurations where time-sharing is actually used in order to accomplish positive and successful domain alterations.

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 workers. At the end of fixed intervals (**quanta**) of length q_a each process forwards this metric tuple to the system scheduler. Over larger quanta, q_s, each system scheduler instance iterates the feedbacks received from all programs p_i and decides on an allotment a_i for each. The decision is based on comparing the current *wasted-cycles* to the total processor cycles, classifying the thread as **inefficient** or **efficient**. In parallel, if $a_i < d_i$ (the allotment is less than desired) the process is **deprived**, otherwise **satisfied**. This last classification speaks to the degree of contention in the system and is independent to the performance of the application.

The decision policy, based on work introduced in [1], is as follows:

- **Inefficient**: overestimation. The process is unable to utilize its domain so the desire is decreased for the next quanta.
- **Efficient and satisfied**: underestimation. The process was allotted its desired resources and successfully utilized them. The desire is increased for the next quanta.

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

[2] Currently there is no support for architecturally heterogeneous systems in the *BIAS* scheduler, since it would require migrating binaries between different architectures

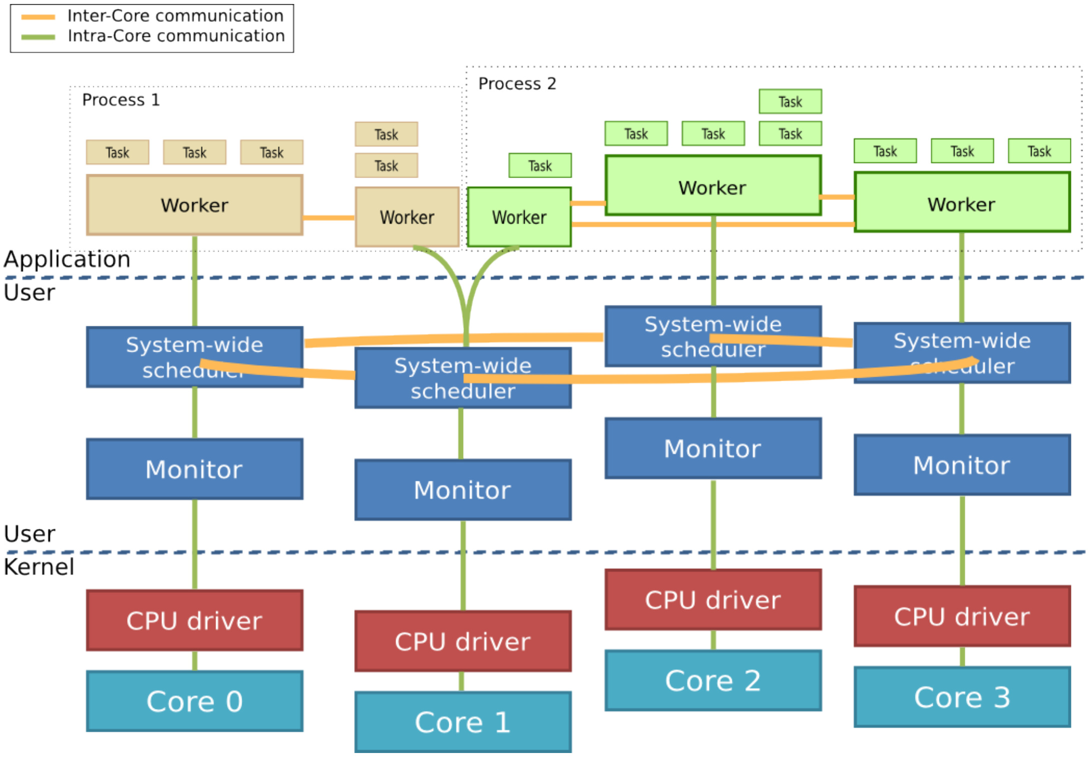

Figure 2. A snapshot of the system, with two processes time-sharing the same core.

- **Efficient and deprived**: balanced. The process was allotted less than desired and utilized them. The desire remains the same for the next quanta because the system can most probably not provide more.

When a program is characterized as *efficient and deprived* the desire for resources remains the same irrespectively of the efficiency of the process due to contention. If other processes become less efficient then resources will be redistributed. If the deprived process gets its desired amount and manages to utilize it, it will then ask for more. This part of the algorithm is trying to balance and space-share the system.

2.4 User-level thread scheduler

At the current stage we are investigating using customized versions of two well-established task-based programming models. Both utilize work-stealing algorithms for task distribution. The first such model is *WOOL* [8], which is extremely fast and efficient in system-resource utilization. Next, *Cilk++* [5] provides sufficient speedup but we are considering it because it provides the basis needed to incorporate malleability of the application's domain. These two programming models although looking very similar in syntax and functionality, 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. Each worker counts its *wasted-cycles*. This clock is stopped on a successful steal and started again after its first failure to steal. Over regular quanta the main thread, that incorporates our augmentations, will iterate the stored statistics and, by employing adaptive work stealing as introduced in [1], calculate its average efficiency and its two worst workers. This information is then sent to the system scheduler.

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 distributes the available processors among them. Such a mechanism would however act as a bottleneck for the system if the system scheduler was one thread on one core.

This is why the system scheduler is implemented as a distributed service. The processor topology is partitioned into sets of cores called **sections**. This partitioning is fixed. 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 second table shows which sections own cores for which processes. This is a simple one-to-many table, mapping processes to cores thus sections; it is called the **participation table**.

The basic principles of this service are:

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. Its process is binded to its local SI. Leader election decides it.
   - 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. Processes can be allotted cores from multiple sections. Local SI might change.
7. Thread schedulers communicate only with their local SI.
8. SIs can request information from other SIs, regarding processes or even specific threads.
9. SIs voluntarily share only the absolute necessary amount of information.
10. The only global knowledge is the *participation table* and the immutable *ownership table*.

Whenever the allotment of cores for a process is changed, there are two events that take place. First a delta is broadcasted to all SIs. Then a simple leader election algorithm is executed for figuring out the local SI to the process. Simple because the leader can be predicted so no information has to be exchanged. 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. If it is changed then the old leader simply forwards its process state data to the new leader.

Over fixed intervals each SI has to decide on a new allotment for each process. It iterates the ones that it knows as *efficient and satisfied* trying to increase its allotment by one. This increase is translated into a decrease for its *inefficient* processes. If no such process can be found it will broadcast a request to other SIs. The first positive reply to arrive is accepted. If none is found then according to a configurable efficiency threshold time-sharing will be allowed on the core that hosts the most inefficient worker of another *efficient and satisfied* process. The actual algorithm is presented in figure 3.

```
allot(SI)
for all $p_{es} \in$ {Efficient and Satisfied} do
    $r \leftarrow getAvailableCore(SI)$
    if $r == NULL$ then
        $r \leftarrow broadcastForAvailableCore(SI)$
    end if
    if $r == NULL$ then
        $r \leftarrow getCoreToTimeshareWith(SI)$
    end if
    $cores(p_{es}) \leftarrow r$
end for

broadcastForAvailableCore(SI)
for all $SI_i \in \{$\{All SI\}$ \setminus SI\}$ do
    $r \leftarrow getAvailableCore(SI_i)$
end for
return $r$

getAvailableCore(SI)
for all $p_{ie} \in$ {Inefficient} do
    if $count(cores(p_{ie})) > 1$ then
        return $mostIneff icientCore(p_{ie})$
    end if
end for

getCoreToTimeshareWith(SI)
$p_{ed} \leftarrow hasMostCores(\{$\{Efficient and Satisfied\}$ \setminus SI\})$
return $mostIneff icientCore(p_{ed})$
```

Figure 3: 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**. To avoid the excess cost of re-creating a worker thread, removed workers are suspend and later resumed if needed. 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 certain points. We have identified two distinct ways to accomplish this.

The first method concerns programming frameworks like *WOOL*, which keep the application state in the stack. The use of the stack 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 is done instead is to **migrate the worker thread** onto a different core, preferably one hosting a *rather inefficient worker thread*. The worker then can continue processing and syncing but not stealing and not spawning new work, eventually being suspended after its tree has been synced. This process shall be called **lazy-suspension**.

In contrast, programming models like *Cilk++*, utilize **continuation passing style (CPS)** cactus stacks, or in other words keep the application state in the heap (shared memory). This allows for actual task migration since any worker can take over the queue of another worker at any point; it is merely a matter of exchanging specific pointers. In this scenario the worker thread can be suspended instantaneous. This method shall be called **immediate-suspension**.

There is no better between the two approaches. 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 involves larger synchronization complexity, while being more effective.

A useful outcome of the load balancing performed by our proposed scheduler is its ability to migrate processes across the processor topology. Eventually the available resources are space-shared in a close to optimal way. Of course for this to be apparent it requires applications with a significant execution time, otherwise there would be a very small window for making any changes. Such an example as observed by our experiments is presented in figure 4.

4. Phase-lock gang scheduling

One important scheduling paradigm introduced with Barrelfish is Phase-Lock Gang scheduling [2]. 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

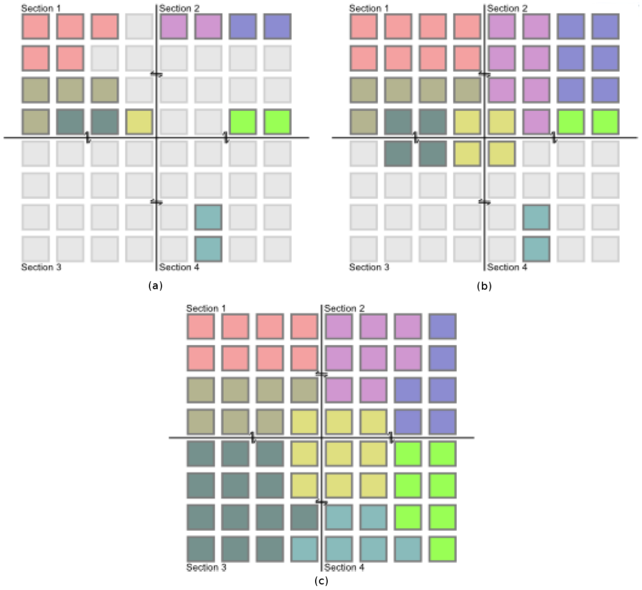

Figure 4: A 48 core processor, divided into four sections. A scenario were multiple processes are initially executed in a small subset of the processor topology (a). For each process, our algorithm will try to assign cores that are local to the initial core were each processes executed on. Given a long execution time, the possibility of contention will be avoided by migrating worker threads onto idle cores (b), eventually achieving maximum utilization of the resources. As seen in (c) the process running on the cores in the center has spread across all sections; section 4 has primary control over it due to owning the majority of its cores.

> without the need for expensive inter-core communication on every dispatch; this feature is being implemented as an extension to the RBED scheduler."

This mechanism works as an extension to the integrated RBED scheduler, accommodating for the various situations where time-sharing resources is inevitable. 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

This project is still ongoing and in this paper we present evaluation of our first naive and unoptimized implementation. To conduct this evaluation we have implemented a set of applications that exhibit diverse types and amounts of parallelism. For these experiments Barrelfish was run using the Qemu simulator and not real hardware. However, since all results are comparative, the same behaviour ought to be similar in other configurations and real hardware.

Given the currently unoptimized API of Barrelfish, there is a very high overhead in bootstrapping both the system scheduler but also an application. This is because of the messaging protocol that has to be followed. Each instance of the system scheduler has to establish a connection with each other for replication, before accepting clients. The same applies for each application which has to find and establish a connection with its local SI before processing starts. For the purposes of these evaluation we are excluding this overhead from the measurements. For all configurations the clock starts right before calling the first task and finishes right after the final syncing.

For each application we have used input data that require processing of at least 30 seconds and up to five minutes. The user-level scheduler quanta, q_u, is fixed at 3 seconds. Each SI will calculates new allotment over fixed quanta, q_s, of 6 seconds. This configuration provides enough time for the application to adapt and for changes to be finalized before a new quanta is up.

The applications used for our evaluation are:

1. **FFT**: Fast Fourier Transformation, with size 120.000.
2. **FIB**: Fibonacci of number 32 and 40.
3. **Knapsack**: Simple packing problem with input of 28 items.
4. **Loop(coarse)**: A simple multiple loop program, where the inner most loop performs long sequential calculations.
5. **Loop(fine)**: A simple multiple loop program, that spawns a big number of fine grained tasks.
6. **nQueens**: The typical nQueens problem over 10x10 and 14x14 boards.
7. **Stress (coarse)**: A simple stress program that floods the system with coarse-grained tasks.
8. **Stress (fine)**: A simple stress program that floods the system with fine-grained tasks.

As presented in figures 5 and 6, executing our testing set of parallel applications shows that the overall cycle count is not greatly affected by the addition of the user-level scheduler and the overhead of the statistics gathering. All applications are run in isolation over an 8 core emulated configuration; all spawn 8 workers, one on each core. It is interesting to notice that highly parallel applications perform better with our scheduler (*BIAS*) as it allows the WOOL runtime to adapt better to Barrelfish while there are minimal wasted cycles.

This adaptation comes from the fact that core 0 hosts most of the system services thus the corresponding worker is quickly marked as inefficient and lowers its tendency to steal work; moreover, it is the thread of worker 0 that also runs our statistics analysis over fixed quanta, which also makes worker 0 steal less tasks. Hence the other workers are less dependent on the unavoidable delay caused by that worker. Contrary to that, coarser grained applications (larger, sequential tasks) are afflicted by this behaviour as the aforementioned overhead cannot be compensated by stealing less tasks.

Figure 7 presents the cycle count of running two applications simultaneously. Specifically, we used two sets of two; the coarse and fine grained loop and stress applications respectively, with and without our scheduler. The results are promising as they show noticeable improvement in performance. Both applications initially spawn 8 workers, one on each core. Without our scheduler, there is obvious contention of resources which affects greatly processing of the more sequential tasks. On the contrary, our scheduler will migrate workers and eventually space-share the system producing a significant increase in performance, especially for the more sequential tasks which are processed uninterrupted as if in isolation.

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 envi-

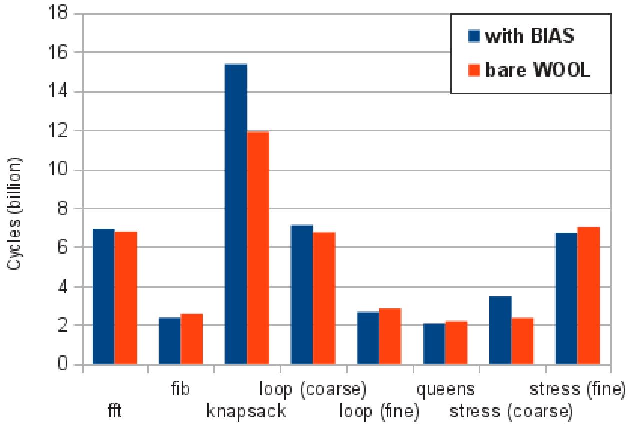

Figure 5: The bars show the average execution cycle count for each application, with our scheduler (BIAS) and without. The applications are run in isolation on 8 cores with 8 workers. In most cases the results are quite similar thus revealing that the overhead of our scheduler to the runtime is minimal.

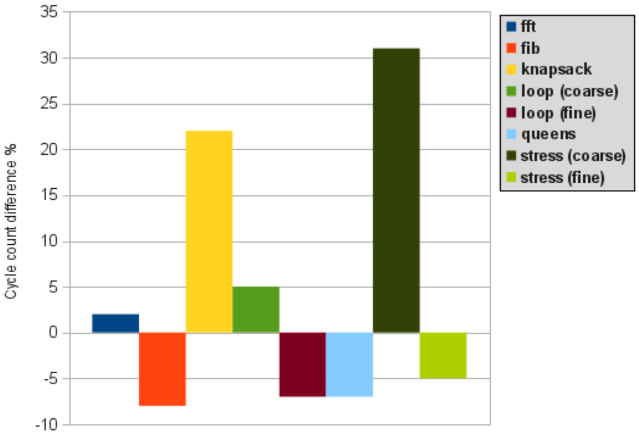

Figure 6: As with figure 5, this is the average percentile difference of cycles. The negative values show that our scheduler allowed better performance. The diversity between each application reveals that the nature of the application, namely the type and amount of parallelism that it exhibits, can affect greatly performance and resource utilization.

ronment, 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.

From our preliminary results it is obvious that a load balancing mechanism is necessary for achieving meaningful performance gains of parallel applications. This project should be considered as a first step to adding such features on Barrelfish. Other non-distributed operating systems like Linux, have provided system-wide load balancing and thread migration for quite some time now, however they lack the portability and scalability properties of Barrelfish. The steady increase of cores for newer processors, and the appearance of multiple new many-core architectures, investigating Barrelfish is surely worthwhile and with our preliminary results seemingly promising.

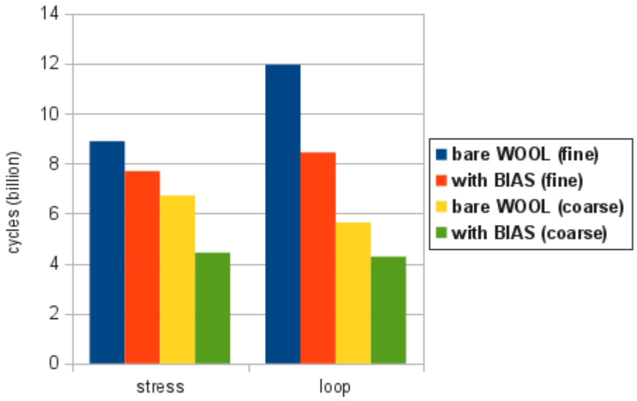

Figure 7: The bars show the average execution cycle count for running two applications simultaneously, with our scheduler (BIAS) and without. A more sequential stress with a more parallel stress and the same for the loop application. The applications are run on 8 cores with 8 workers initially each. In all cases, our scheduler allowed significant improvement in performance by space-sharing the system.

The split nature of our proposed scheduler allows it to be mapped and adapted in configurations and systems which combine multiple, inherently distinct layers dividing the scheduling logic with the scheduled entity. Barrelfish OS is just one example and a rather fruitful experimental testbed.

7. Future work

Our immediate intent, is to extend our evaluation of the system with more complicated configurations and proper benchmarking tools. Moreover, we plan to evaluate it over real hardware. Target architectures are the Intel i7, a 4x12-core AMD opteron system and Tile64Pro. Porting Barrelfish to the Tile64Pro architecture is currently under way but not yet complete.

Long term goals focus on highly optimizing the scheduler's platform and the corresponding API of the OS, while expanding its features:

- Experiment with other task-based programming models like Cilk++, as well as non-work-stealing like OpenMP.
- Add locality-awareness to the system scheduler for better selection of workers to suspend in relation to the process that requires the released resources.
- Make use of processor characteristics as criteria on the decision of core allotment for the deployment on heterogeneous architectures.
- Handle the absence of shared-memory support by the underlying architecture.

8. Acknowledgements

I would like to thank Microsoft research for sponsoring this project, as well as everyone contributing to Barrelfish through source core or the mailing list. Finally a special thanks ought to be given to the Swedish Institute of Computer Science (SICS).

References

[1] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26(3):1–32, Sept. 2008. ISSN 07342071. doi: 10.1145/1394441.1394443.

[2] A. Baumann, R. Isaacs, and T. Harris. Design Principles for End-to-End Multicore Schedulers Context : Barrelfish Multikernel operating system. *Group*.

[3] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. 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.

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

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

[6] S. 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. doi: 10.1109/REAL.2003.1253287.

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

[9] D. Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). *IBM Research Report RC19790 (87657) 2nd Revision*, 16(1):104–113, May 1997. doi: 10.1145/1007771.55608.