Scheduling of Operating System Services

Stefan Bonfert
Ulm University, Institute of Information Resource Management
Ulm, Germany
stefan.bonfert@uni-ulm.de

1 INTRODUCTION

While modern applications are often multithreaded, the manual implementation of concurrent programs is inconvenient and error-prone. One way to ease the implementation of such programs is to specify them as a set of tasks in data- or workflow graphs. These graphs describe the individual tasks, causal dependencies and data dependencies between them. Once all dependencies of one task are satisfied, it can be executed. Thereby, the order of execution is decoupled from the implementation of the program, which enables the underlying scheduler to exploit parallelism more efficiently.

For its execution, a program relies on services offered by the operating system like, e.g., memory management, thread management or I/O. In order to utilize these services, applications use system calls. These transfer control from user space (application) to kernel space (operating system). Traditionally, these system calls are executed synchronously on the CPU core, where the system call was invoked by the application. This execution model comes with two major drawbacks: (a) The invoking task is blocked while the system call is executed and (b) Due to the execution of different types of system calls on the same CPU core, caches are overwritten when switching between them. Modern microkernels like Fiasco [1] and MyThOS [2] use message passing as a communication method between different kernel instances running on different CPU cores or even different nodes in a data center. Therefore, they are able to support system call forwarding, which solves the problems mentioned before and works as follows: When an application uses a system call, the local CPU core jumps to kernel mode and executed the respective system call handler. This handler function forwards the system call to another suitable OS instance running on a different CPU core and returns control to the application. Thereby, blocking is avoided as much as possible. Afterwards, the result of the system call can be fetched by the application asynchronously. By routing all system calls of a specific type to one or few dedicated CPU cores, the caches on these cores stay populated with related data, which even helps to speed up the execution of the system calls themselves.

However, it is currently unclear which cores should process which system calls and thereby offer a specific service. Services like memory management can be executed by a single core or, in the case of distributed shared memory, by a central node in the cluster. This strategy may lead to high load on that core and therefore high waiting times for the system call's result depending on the behavior of the application and the size of the computing cluster. This can be solved by replication of the respective service, which in turn leads to communication overhead due to synchronization and consistency overhead between the service instances. Therefore, the operating system requires a service placement system to determine the optimal placement and replication strategy for the service instances for each system call. It should be able to determine a mapping of service instances (processing a specific system call) to CPU cores. This decision should build upon information made available from the application and collected during its execution.

2 APPROACH

In this PhD thesis both a task mapping infrastructure (task scheduler) and a system for OS service placement (service scheduler) are developed. Due to different coherency levels (e.g. shared cache, shared memory, different nodes) and heterogeneous hardware (e.g. CPU, GPU), a hierarchical task scheduler was selected, where the hierarchy levels correspond to different coherency levels of the computing infrastructure, i.e., cluster, node, shared memory, shared cache. This task scheduler assigns tasks from an application's flow graph to computational resources depending on (a) location of the input data, (b) its availability, (c) the expected runtime of the task and (d) the location of required operating system services.

To determine whether a task is ready for execution, its data dependencies have to be evaluated. From the application's perspective this is a difficult task, since the location of data is not transparent to the application. Therefore, the task scheduler developed in this thesis will be integrated into the operating system. It therefore can more efficiently check dependencies, since data mapping and memory layout are available.

The dependencies of tasks upon specific operating system services (i.e. the information weather they may be called) are defined explicitly in the task description and can be extracted by the compiler. This information is used by the task scheduler for its initial placement decision.

In many fields, e.g. HPC, tasks are executed iteratively. In the majority of iterations, they exhibit similar behavior, e.g. a task may always reserve memory in the beginning and release it before terminating. Therefore, the calls of tasks to OS services can be predicted in terms of patterns. In order to identify these patterns all system calls have to be monitored. For this purpose, each task is assigned an ID either by the developer or the compiler. If two tasks have the same ID, they will execute the same function. This ID is used by each service instance to record the number of accesses per task ID over a given time frame. Using this information, the service scheduler can then determine affinities between services and task IDs. These are then used to place new tasks close to the services they will frequently use or replicate services to be more locally available to such tasks. In this work, tasks are considered to behave similarly across many executions. Therefore, system call patterns do not change rapidly, but they can be better approximated by observing additional executions.

If the response time of a service instance is too high, it may be either overloaded or located too far from the tasks using it. In this case the service scheduler has to consider different options. While the replication of this service adds synchronization and consistency overhead, the migration of the instance closer to the tasks using

it may avoid this overhead. However, depending on the behavior of the application's tasks, replication may improve the overall performance of the system. The costs of both decisions, including reconfiguration cost, are evaluated by the service scheduler in order to find an optimal solution.

3 CHALLENGES

Although, many tasks, e.g. in HPC, exhibit similar runtime over multiple invocations it may vary between iterations or change entirely based on input data. On the other hand, the task scheduler requires information about the workload of a task to be able to schedule successive tasks accordingly. Therefore, a suitable estimation of a task's runtime has to be found. This could be, e.g., exponential smoothing of previous runtimes. Based on input data tasks may exhibit entirely different behavior. However, this effect is expected to be negligible, since it often occurs at loop borders, which only make up a small fraction of the invocations.

Similar considerations have to be taken for the system call behavior of tasks. The frequency of system calls may evolve over time and change rapidly in some invocations. This is again compensated by exponential smoothing.

One major challenge is the mutual dependency of the two schedulers. Given an operating system configuration with fixed service instances, the task scheduler can determine an optimal task schedule based on this OS configuration. In turn, the service scheduler can determine an optimal service placement, depending on a fixed task distribution. However, when simultaneously scheduling tasks and services, the schedulers interact with each other and influence each others' decisions. These effects will be studied and quantified within this thesis and the schedulers will be designed to avoid unstable behavior.

4 CONCLUSION

In this thesis, scheduling strategies for both application tasks (depending on data location and availability) and operating system services will be developed. The cost of reconfiguration in both task and service placement will be evaluated and will serve as a metric for the scheduler. Thereby, local and global performance of the system should be optimized.

REFERENCES

[1] B. Döbel, H. Härtig, and M. Engel. Operating System Support for Redundant Multithreading. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 83–92, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1425-1. doi: 10.1145/2380356.2380375. URL http://doi.acm.org/10.1145/2380356.2380375.

[2] R. Rotta, J. Nolte, V. Nikolov, L. Schubert, S. Bonfert, and S. Wesner. MyThOS - Scalable OS Design for Extremely Parallel Applications. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 1165–1172, 2016. doi: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0179.