WHITEPAPER

IMPROVING EXECUTION PREDICTABILITY ON LINUX WITH SLX
## Contents

1 Introduction .................................................. 1

2 The Linux Scheduler ........................................... 2

3 Using SLX ....................................................... 4
   3.1 Dealing with Non-Determinism ......................... 4
   3.2 Dealing with Unpredictability ....................... 4

4 Multiple Applications .......................................... 5

5 Conclusion ..................................................... 8

6 Acknowledgement ............................................. 8
Executive Summary

For many applications, predictability and determinism are often times more desirable than raw performance. This is especially true in emerging markets, like cyber-physical systems or the internet-of-things. For many practical reasons, however, most engineers rely on Linux, which in multicore systems is usually neither predictable nor deterministic. This whitepaper analyzes the predictability of executing a task on an off-the-shelf heterogeneous system running Linux. By using SLX, the execution model of applications provides provably deterministic behavior. Similarly, the whitepaper shows how using SLX can improve predictability by several orders of magnitude, by pinning threads to specific cores, while keeping competitive performance compared to execution on plain Linux.

1 Introduction

When designing systems that interact with the environment, engineers usually do not strive to get the maximal performance out of the device. Instead, soft or hard real-time constraints usually have to be met for the device to function properly. In these cases, predictability and determinism are usually more valuable than raw performance, the latter especially for the correct functioning of the device. With the ever-increasing market for Cyber-Physical Systems (CPS), as well as the ever more demanding requirements set to these, it is crucial for companies to optimize time to market in these domains to remain competitive. A prominent example of increasing demands is the requirement of most modern devices to be interconnected using the Internet Protocol, also known as the Internet-of-Things (IoT). This requires various services and protocols to be operated alongside the original function of the device, and usually demands the device to react interactively to additional devices communicating with it.

Traditionally, real-time systems have a particularly long time to market. Formal verification and worst case execution time analysis are time consuming and very inflexible. Real-time operating systems are complex, and a full system design has to be done for any new product. On the other hand, when a quick development cycle is required, engineers often rely on technologies that are widespread and well-tested. A prime ex-
ample of this is Linux, which runs on almost all off-the-shelf hardware products and is time-tested, robust and simple to use. While scheduling in Linux is very varied and configurable, the standard Linux scheduler is not particularly predictable. In general, Linux does not provide the same guarantees of predictability and determinism as most real-time operating systems do.

With the emerging markets in CPS and IoT, engineers need to design products which behave in a deterministic and predictable fashion while relying on simple and tested technologies, like Linux. To make ends meet, new technologies are required that allow designers and engineers to bridge this gap. In this whitepaper, we show benchmarks measuring the unpredictability of Linux, and how this can be significantly improved by using SLX to obtain deterministic, predictable execution with good performance on off-the-shelf hardware running Linux.

2 The Linux Scheduler

Linux is a widespread operating system, used today across many domains. In this section we argue why this operating system, and others alike, are not a good choice for applications that require time-predictability and deterministic execution.

The Linux kernel can, in principle, use different schedulers or scheduling policies. Since version 2.6.23, released almost a decade ago at the time of this writing, the default scheduler in Linux is the “Completely Fair Scheduler” (CFS). CFS aims to give all executing processes a fair share of processor time, while maximizing CPU utilization. This is desirable for general purpose computing where responsiveness in the presence of dynamic loads is key. However, as we will show, CFS does not provide good time predictability, i.e., the execution time of several instances of the same application displays a high variance. This is especially true in scenarios with dynamic, changing...
system loads. Thus, while CFS is designed to produce the best performance in dynamic scenarios, it does so by sacrificing predictability, which is particularly noticeable in these cases.

In order to assess the predictability of the system, we present a case-study done by researchers from TU Dresden. In it, we evaluate two different benchmarks on a modern off-the-shelf multicore architecture, the Hardkernel Odroid XU3. The system includes an Exynos 5422 big.LITTLE chip with four Cortex-A15 cores and four Cortex-A7. Additionally, it features 2 GB of LPDDR3 RAM clocked at 933 MHz. Selected benchmarks include a filter application for stereo audio signals (Audio Filter - AF) and a multiple-input multiple-output (MIMO) transceiver for wireless telecommunications.

The two benchmarks were compiled using SLX, but without any support for thread pinning. Instead, we let the different posix threads of the application be scheduled completely by CFS. Using a temporary filesystem to mitigate noise from I/O effects, both benchmarks were executed, each fifty times, and the timings measured for the different executions. Similarly, using the on-board energy sensors featured on the Odroid XU3 the energy of the system was measured. Connected via the I2C bus, these INA-231 sensors measure the energy at the voltage regulators of individual components. We report aggregated values for the energy.

Figure 2.1: Variance in performance and energy consumption of Linux CFS.
Figure 2.1 shows the results of different timings by CFS scheduling on Linux. The results reveal that, indeed, CFS can be very unpredictable. The relative variance of execution times reaches up to 30%. We see a similar behavior regarding energy consumption. Additionally, CFS potentially always uses all cores for execution, independent of how many are needed. It is clear that any application which requires predictable execution times cannot rely on Linux CFS with posix threads alone for scheduling.

3 Using SLX

In this section we explore how SLX can greatly mitigate two problems of Linux CFS, predictability and non-determinism, while keeping a competitive performance.

3.1 Dealing with Non-Determinism

Apart from the variability induced by the kernel, parallel applications are written today using posix processes or threads (or abstractions thereof like OpenMP). These parallel abstractions are known to be non-deterministic [4]. In a multi-threaded application, the scheduling of threads depends on intangible variables of the system’s state, that cannot be accounted for or predicted at compile-time. This ranges from abstract states like which memory lines are currently in the cache, to minute temperature differences which could have an effect on the execution. This is also true for Linux with CFS, as a thread model with a dynamic scheduler. In particular, it implies that the programmer cannot have any guarantees of the order of execution, and makes debugging orders of magnitude more difficult.

A solution for this is not using threads directly, but instead to use a deterministic programming model [4]. SLX supports a programming model based on the Kahn Process Network semantics, which is provably deterministic, yet extremely flexible and expressive [3]. Thus, an application designed using SLX will behave in a deterministic fashion, even when executing on the highly non-deterministic posix thread model of Linux.

3.2 Dealing with Unpredictability

Executing a multi-threaded application on a modern multicore system adds a significant depth to the compilation problem. In order to produce efficient code, the compiler has to consider several additional constraints, not present in a single-threaded scenario. Data dependencies between threads, different execution times on different processing elements, and complex communication patterns with different latencies in the memory...
subsystem; all these play a crucial role in ensuring a multi-threaded application will execute efficiently. SLX includes a powerful analysis and performance estimation suite. Using it, it can estimate the performance of your application on many different configurations, regarding all these additional constraints. Then, SLX uses these estimations to fuel novel algorithms, which can quickly find the best possible configuration, selecting the ideal core for every thread. This is also called a mapping.

By pinning cores to threads, the Linux scheduler loses flexibility. However, with the detailed analysis outlined above, this can in fact be advantageous. By pinning threads to cores, SLX prevents unnecessary thread migrations, which flush the caches and slow down the application, or execute the less-critical threads on the most performant cores.

With the same setup as in Section 2, we show the comparison of Linux using CFS to execute a single application, compared to a Mapping calculated using the SLX Tool suite. As a constraint on the resources, this mapping uses only two of the eight cores, a “big” and a “little” one. In this way, restricting the used resources, the application can be used alongside a system with a higher load without interfering with applications running on the remaining cores. Since CFS schedules using all cores, it is clear that it will outperform the SLX mapping. To give a fair comparison, we also report values measured by letting CFS schedule all threads, but restricting the execution to the same two cores on the mapping. We call this strategy “Core Pinning”, for the purposes of this analysis.

Figure 3.1 shows the results of the comparison. In the figure we can see how, of course, using all eight cores yields better performance than only using two, albeit at the cost of predictability. While we already see a significant improvement in predictability by restricting the resources used, with the “Core Pinning” strategy, we see that mappings obtained by SLX yield competitive performances, while being significantly more predictable than CFS with the “Core Pinning” strategy.

4 Multiple Applications

While Linux is already unpredictable when executing a single multi-threaded application, this problem is only aggravated when several applications compete for the system’s resources. Since the scheduler does not understand the interaction of threads within and between different applications, it will schedule threads where it finds available resources. While this can result in a very efficient utilization of resources, it can also waste precious compute time of threads unnecessarily waiting, purge caches constantly, and let different applications interfere with each other. Altogether, this results in even more unpredictable behavior.

On the other hand, multi-application scenarios are becoming the norm in modern cyber-physical systems and other embedded devices. Versatile embedded systems
need to be able to adapt to different workloads, while keeping predictable and correct behaviors, which often times also includes meeting timing deadlines.

We have seen how mappings, as generated by SLX, significantly improve the predictability of the execution of a single application. To deal with multi-application scenarios, however, mappings are not enough. It is not tractable to consider every possible system workload and calculate mappings for those, and if single-application mappings are used, unintended resource collisions will set back all benefits of the mapping. To deal with this, we need a method to transform mappings in such a way that keeps the benefits of mappings and allows to dynamically execute multiple applications using different resources.

In collaboration with researchers from TU Dresden, we tested bleeding-edge methods for transforming mappings to execute in multi-application scenarios with SLX. We did so by using the Transitive Efficient Template Run-time System [1] (TETRiS) developed at TU Dresden. TETRiS understands the underlying target architecture, and uses this knowledge to transform a mapping in a way that guarantees the same per-core performance and inter-core communication patterns. These transformations are automatically derived from mathematical models obtained from the architecture description, leveraging inherent symmetries of the problem by using novel algorithms and mathematical methods [2].
Consider the example depicted in Figure 4.1. We want to execute the application depicted on the figure, represented by the blue graph. Using SLX, we calculate a mapping with a particular goal, e.g. optimal execution time. However, at runtime, the system has an existing load, using four cores, as depicted in the figure. By giving this information to TETRiS, it understands how to transform the mapping from SLX and schedules a mapping that will give the same performance, but uses the resources available. In this case, the transformation can be seen as a rotation by $180^\circ$ combined with a translation. The transformations do not always have such a geometric interpretation, but they still yield the same performance [1].

Using TETRiS, we evaluated the predictability of the execution with the same setup as in the previous sections. Instead of single applications, however, we evaluated two different multi-application scenarios. The first one features four separate instances of the Audio Filter benchmark, while the second one uses two instances of the Audio Filter benchmark, and two instances of the MIMO benchmark. The results can be seen in Figures 4.2 and 4.3. We see how CFS is again very unpredictable, and the Core Pinning strategy, while an improvement over CFS, is also still quite unpredictable. Additionally, since this experiment features a full system load, the advantage of having more cores from CFS is completely mitigated. The mappings obtained with SLX with TETRiS perform just as good as CFS in average, while being significantly more predictable. In fact, the variance of execution times in CFS in this experiment was a factor of 510 higher than that of the mappings obtained by SLX.
5 Conclusion

We have shown how using Linux with CFS, while convenient, leads to non-deterministic and very unpredictable results. Both issues can be addressed by using SLX, which uses a deterministic programming model and novel algorithms to estimate and find optimal mappings of threads to cores. Measuring on the Hardkernel Odroid XU3 we showed how, indeed, the predictability can be significantly improved by using SLX, reaching a variance a factor of $510$ lower in the execution time of applications, compared to Linux CFS. Especially in multi-application scenarios, with dynamic and changing system loads, SLX with novel methods from researchers at TU Dresden can keep a predictable and efficient execution.
6 Acknowledgement

Silexica would like to thank Andrés Goens and Prof. Jerónimo Castrillón from the Chair for Compiler Construction at the TU Dresden for sharing their research results, providing invaluable insights and supporting the creation of this document.

Bibliography


