Evaluation of parallelism techniques on embedded multi-core platforms

Lucas Jenß
Betreuung: Prof. Dr. B. Schwarz

Hamburg University of Applied Sciences
lucasandreas.jenss@haw-hamburg.de

Contents

1. Introduction and Motivation

Embedded platforms and applications are being used throughout many industries, including telecommunication, robotics, medical and automotive (Wang et al., 2013). Campeanu (2013) states that nowadays it is increasingly common to build embedded systems on heterogenous platforms, containing multiple computational units such as multi-core CPUs, GPUs and FPGAs. If all these components are used to the full extent of their capabilities, such a setup enables better performance than could be previously achieved on embedded platforms.

This is an important shift, since applications running on these platforms become more and more complex, and their performance requirements have to be satisfied. In the past, not only on embedded platforms, this was commonly achieved by boosting processor clock rates. But due to both intractable physical limitations as well as practical engineering considerations and requirements (i.e. low power consumption for a battery-powered devices), increasing processor clock rates is no longer a feasible approach (Cantrill and Bonwick, 2008).

The performance gains possible on heterogenous embedded platforms are a result of the hardware-level parallelism that is achievable on these platforms. But unfortunately, exploiting these parallelism features still is a considerable challenge, given the tools and programming concepts widely in use today (Wang et al., 2013; Cantrill and Bonwick, 2008).

Because of the mentioned increase in complexity and performance requirements in embedded applications, and considering that clock rates can't be expected to increase significantly anymore, research in the area of concurrent applications has taken an important place in the computer science research community. As a result, an increasing number of publications related to concurrency have been presented over the last 7 years (compare Figure 1).

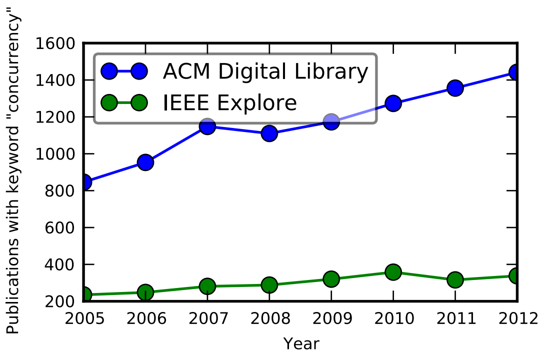

Figure 1: A strong increase in publications related to concurrency can be observed over the last years. These numbers were researched from the IEEE Xplore (2013) digital library as well as the ACM (2013) digital library.

In context of the FAUST Project (2013) at the Hamburg University of Applied Sciences, a specific embedded application in need of parallelization is an obstacle detection system in cooperation with a lane guiding control, for which an overview is given in section 2. Evaluating the distance data measured by a laser scanner and the image stream providing extracted road marking parameters constitues a dataflow parallelization problem, for which suitable algorithms must be researched and compared. Therefore, this essay focuses on the literature evaluation of techniques that allow for the exploitation of potential sources of parallelism, especially those which are suitable for dataflow problems (section 4). Said evaluation is done in context of a modern multiprocessor system-on-chip platform (MPSoC), the Xilinx Zynq 7000.

Additionally, the current lack of widely used tools and languages crafted for use in concurrent environments is addressed by giving a short overview of the "Go" programming language (section 5). It was conceived, among other things, to allow for a highly productive development environment for parallel applications. The hypothesis is thus, that by using the Go programming language to develop parallel applications on embedded platforms, one can increase developer productiveness by allowing the language to take care of tasks such as thread scheduling and memory management. Given that said features come at the cost of performance, their impact on embedded systems must be evaluated (subsection 5.1).

Given that the FAUST project focuses on MPSoC platforms for implementation, the evaluation is confined to the area of Symmetric Multiprocessor (SMP) systems, i.e. those having multiple processors or processing cores with shared memory. Other parallelism concepts for architectures such as Distributed Memory Access or Nonuniform Memory Access will not be discussed.

2. Platform and problem description

This section covers a short overview of the platform currently being used for embedded application development in the FAUST Project (2013), the Xilinx Zynq 7000, along with the applications running on that platform (visualized in Figure 2). This overview is given to establish a context for the subsequent sections, and to give an idea of recent work. At present, there are two major subsystems that need to run in parallel on this platform, which are part of a technology field trial of advanced driver assistance systems:

1. An obstacle detection system (subsection 2.1)
2. A lane guiding control system (subsection 2.2)

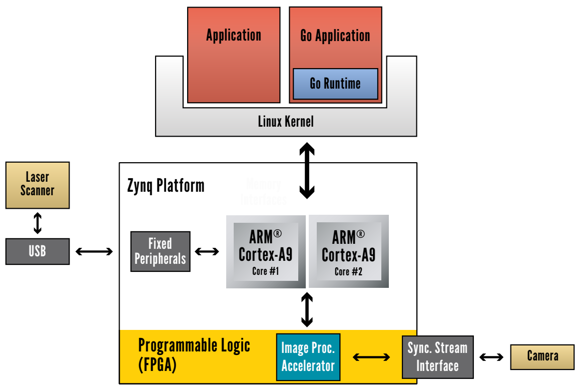

Figure 2: Overview of the platform being used, as well as its subsystems.

The Xilinx Zynq 7000 architecture is composed of a dual-core ARM Cortex-A9 CPU and an on-chip programmable logic unit (FPGA). The ARM Cortex-A9 additionally contains the ARM NEON general-purpose SIMD engine (Xilinx, 2013), for which an overview will be given in subsection 4.1.

2.1. Obstacle detection system

This laserscanner-based obstacle detection system was realized by Jestel (2013), providing an implementation in the Java programming language running on the Android 4 operating system. Because of the fact that the Zynq 7000 platform was not yet available when development on the system started, it is currently running on the "Open Multimedia Application Platform 4430" (OMAP4), which features similar specifications to the Zynq platform, most importantly a dual-core Cortex-A9 CPU, but lacks the integration of an FPGA.

In his implementation, Jestel (2013) employs two worker threads assigned to each of the CPU cores, and sets them to maximum priority for the operating system scheduler. Each of the worker threads alternate at receiving and evaluating the data received from the laserscanner. The synchronisation is realized through software mutexes (compare Figure 3).

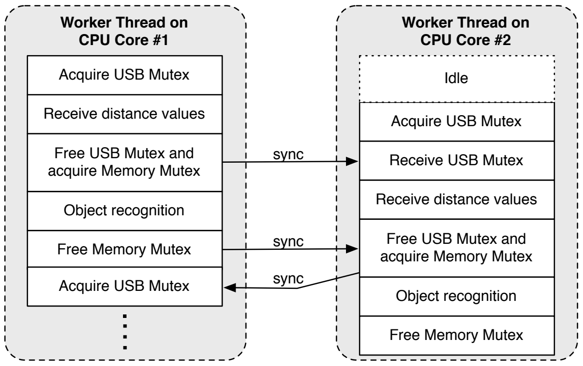

Figure 3: Parallelization concept as used by Jestel (2013) in his obstacle detection system implementation.

2.2. Lane guiding control system

The previously developed System-on-Chip Autonomous Vehicle architecture devised by Johannsen (2013) was ported to the Zynq 7000 platform by Andresen (2013). When porting the architecture he focused on distribution of computational tasks between the available CPU cores and the programmable logic (FPGA), thus making optimal use of the available resources in order to accommodate additional subsystems in the future (such as the obstacle detection system).

The resulting system uses the FPGA for processing of the raw image data received from a 60 FPS camera with a resolution of 640x480px. The output data from the FPGA is then evaluated by one of the available CPU cores, which then calculates the steering parameters to keep the vehicle driving in its lane. The other core manages the computations executed on the FPGA plus any operating system tasks that may arise.

The developed system runs on a Linux Kernel v3.2, and it should be noted that the system's timing constraints for the lane guidance were kept despite the fact that no real-time patch (e.g. `CONFIG PREEMPT RT`) had been applied to the kernel (the scheduler was set to `SCHED FIFO`). It is reasonable to assume that this was possible due to the relatively low framerate and resolution of the camera, which resulted in an low overall system load. With increasing load, however, it might not be possible to keep timing constraints without either applying a real-time patch to the kernel or using a real-time operating system (RTOS).

3. Parallel computing architectures

A system for classification of parallel computing architectures is outlined in this section, to provide context for the subsequently explained parallelism techniques. Flynn (1972), in what is often referred to as "Flynn's Taxonomy", states that there are four broad categories of computer architectures:

SISD	Single instruction, single data
MISD	Multiple instruction, single data
SIMD	Single instruction, multiple data
MIMD	Multiple instruction, multiple data

SISD is what can be found in most single-core computers, where a single stream of input data is processed by a single stream of instructions. SISD is not further discussed since it has no parallel elements, and MISD is completely ommited because there are no well-known systems that fit into this category (Mattson et al., 2004).

In this essay, the focus is on SIMD and MIMD architectures, which offer a potentially high degree of parallel execution.

3.1. Multiple Instruction, Multiple Data

The MIMD category in Flynn's taxonmy, as depicted in Figure 4, features multiple control-units which provide instructions to a number of processors, each with their own data input stream. Additionally, the MIMD processing units may be interconnected. This category is the most general, and applies to all multi-processor system currently available (Mattson et al., 2004).

3.2. Single Instruction, Multiple Data

Given a single stream of instructions and multiple streams of data, an SIMD architecture is one that applies each instruction to all of the data streams **in parallel**. This process is visualized in Figure 5, where there is a single control unit providing the processors with instructions. For every instruction, every processor takes an item out of its own input stream and applies the instruction.

As such, SIMD architectures are a good fit for specialized applications that consist of high data parallelism along with little interprocess communication. As an example, in digital signal processing, e.g. decoding a video stream, the situation is common that the same operations need to be applied continously to a large stream of data (in case of a 1080p high-quality H.264 movie, the bitrate is usually above 17Mbit/s). As such, SIMD instruction sets have become a generic feature of most high-end processors because of the possible performance improvements that they can provide (Jang et al., 2011).

A Graphics Processing Unit (GPU) can also be seen as a heavily parallel SIMD system. GPUs usually consist of an array of multiprocessors, each in turn consisting of multiple processing

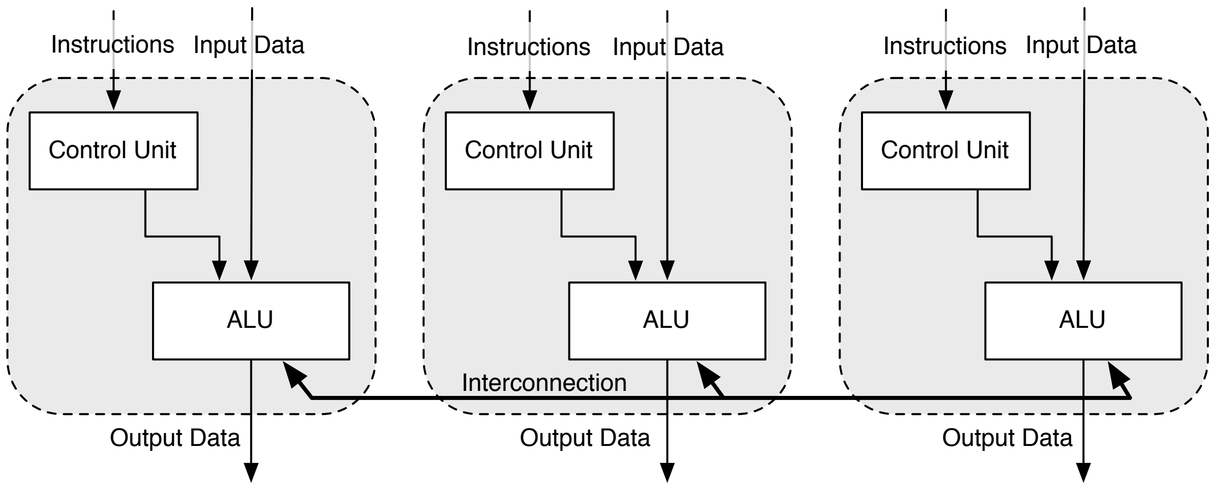

Figure 4: Visualization of a Multiple instruction, multiple data architecture. There are three different processing units, each featuring its own control unit and processor. All processing units have disjoint instruction and input data streams. The processors may be interconnected among each other.

units (ALUs), which all share a single control unit (exactly as depicted in Figure 5). Each of these ALUs is then able to execute instructions provided by the control unit, potentially providing a many-fold performance increase.

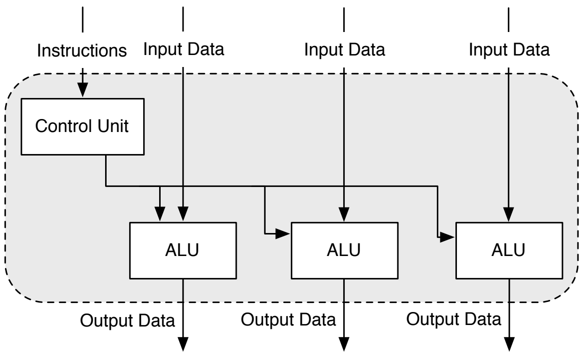

Figure 5: Visualization of a Single instruction, multiple data architecture, containing three processing units. Heavily inspired by Mattson et al. (2004)

4. Parallelization Techniques

In this section, selected techniques will be explained in context of the concrete problem at hand, that is the parallelization of the obstacle detection system on an ARM Cortex-A9 CPU. Since there are a multitude of techniques, and not enough time available to evaluate all of them, the selection was mostly based on the information provided by Mattson et al. (2004). If available, current research based on these techniques will be highlighted.

Perhaps say something about exploitable concurrency

4.1. ARM NEON SIMD

The ARM NEON extension is a general-purpose 128-bit SIMD architecture available in ARMv7. It is included in the ARM Cortex-A9 processors that can be found on the Zynq 7000 (Jang et al., 2011).

The NEON extension is composed of 32 64-bit registers, which can alternatively be used as 16 128-bit registers. In NEON terminology, a register is a vector of elements of the same data-type (possible data-types are 8-bit to 64-bit signed and unsigned integers as well as single precision floats). Every instruction given to the NEON engine is then performed on all elements of one or more such vectors. A "lane" is the part of all registers which shares the same operation. This process is visualized in Figure 6.

Even though the NEON engine is realized as part of the ARM core, it has a completely independent execution pipeline and a register bank that is separate from the ARM core register bank (Jang et al., 2011). As a result, it is possible to signal the ARM core memory system to pre-load memory (using the PLD instruction, see ARM Ltd. (2013)) while the SIMD engine is computing, so that the data for the next SIMD operation will

Add a few standard operations, and maybe a short example

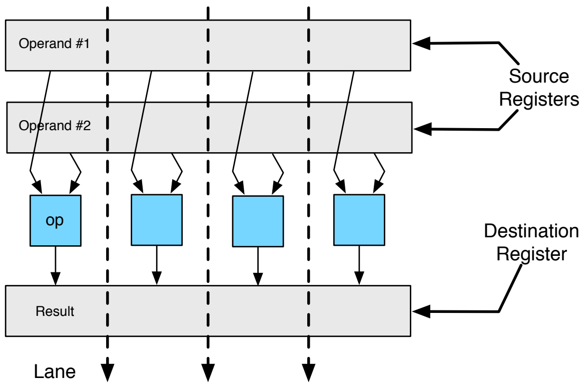

Figure 6: Visualization of the NEON SIMD data flow. Assuming 64-bit source registers, each of the operands is 16-bit wide. The blue squares represent the operations applied to the registers. Four operations are applied, and thus there are four lanes.

```
void f(unsigned short *pDst, short *pSrc,
       short coeff, short intercept,
       unsigned int count) {
    int res;
    do {
        res = *pSrc++ * coeff + intercept;
        if (res & 0x80) res += 256;
        res >>= 8;
        if (res < 0) res = 0;
        if (res>0xffff) res = 0xffff;
        *pDst++ = (unsigned short) res;
    } while (--count);
}
```

Listing 1: Algorithm specifically crafted for a performance comparison between C code and the NEON engine. Source: NEON Alchemy Lab (2013)

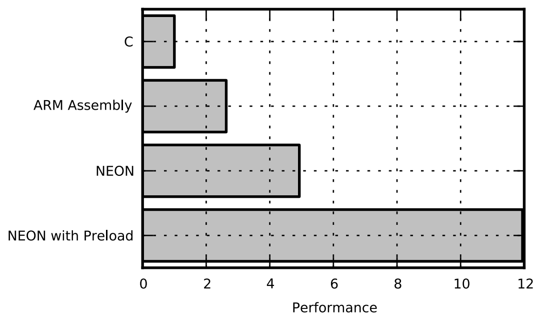

Figure 7: Results of a performance comparison conducted by NEON Alchemy Lab (2013) (not associated with ARM Ltd.).

already be available in the cache. This is important since cache misses can impose a significant performance overhead (Zang and Gordon-Ross, 2013), because the processor can not perform useful operations while loading memory into cache.

A performance comparison conducted by NEON Alchemy Lab (2013) (not associated with ARM Ltd.) shows that, in case of the NEON engine, pre-loading subsequently needed data into the caches while performing SIMD operations can increase performance by a factor of 2.4 for an example algorithm containing multiplication, addition and divison with rounding (data size about 4MiB). It should be noted that this is the performance increase compared to an implementation that already uses the NEON engine. When comparing the NEON implementation with preloading to an implementation in pure C, the performance increase factor is 11.946 (compare Figure 7). The algorithm, which does not serve a particular purpose other than benchmarking, is shown in Listing 1.

4.2. Pipeline Pattern

The pipeline pattern is a technique that can be used to parallelize problems composed of a sequence of operations which have to be applied to a sequence of data elements, also called a "dataflow problem" (Mattson et al., 2004). Pipelining techniques are widely used throughout the computing industry: at a very low level, virtually every modern microprocessors is equipped with an instruction pipeline to increase performance (Finlayson et al., 2013). Higher up in the abstraction hierarchy, UNIX applications and shell scripts often rely on "pipes" as an interprocess communication mechanism. When considering application development, the pipeline pattern is often used because it allows the transformation of a sequential problem to a parallel solution with relatively low effort, especially without a full rewrite of a codebase (Preud'Homme et al., 2012). The pipeline pattern is a MIMD technique.

An example of a dataflow problem is the application of a sequence of image filters to a sequence of source images (like the one depicted in Figure 8). While it isn't usually possible to apply multiple of these image filters to the same image at the same time, because they requires the output from the previous filter, each of them can be applied to **different images** concurrently.

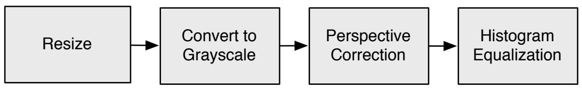

Figure 8: Example of a sequence of image filters. These filters need to be applied sequentally to every image in order to get the correct result.

The pipelining technique then consists of dividing these sequential operations (e.g. image filters) into several pipeline stages. Each of the pipeline stages only ever processes a single data element (e.g. image) at the same time, but all pipeline stages work on separate images concurrently, as shown in Figure 9. This requires the pipeline stages to be interconnected, since every stage must pass on the computed data to the next stage and receive the next data to compute from the previous stage. The implementation of a pipeline stage always adheres to the same structure:

1. Receive data from the previous stage
2. Perform operations on the received data
3. Pass the data on to the next stage

In order for a pipelined computation to have positive performance impact, some important constraints must be kept:

1. The time it takes to compute each stage's result must be much higher than the time it takes to pass the data on to the next stage. If this is not the case, the communication overhead introduced by the pipeline stages will outweigh the benefit of computing the stages concurrently (Mattson et al., 2004).
2. Optimally, each stage takes about the same time to compute. If any pipeline stage completes its calculation much faster, it will be stalled by the subsequent stages, because it cannot pass on the computed data. On the other hand, if any pipeline stage is much slower than the others, subsequent stages will often have to wait for data to arrive, thus wasting computing time.

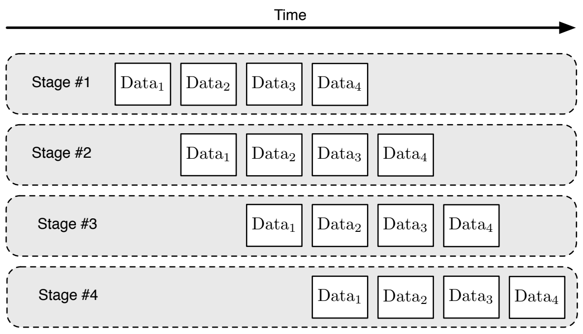

Figure 9: A four stage pipeline processing four data elements. In the first time frame, only the first pipeline stage is active, processing Data$_1$ (all other stages are idle). In the second time frame, Data$_2$ is being processed in stage one, Data$_1$ in stage two, and so on.

Recent research in this area includes the concept of "pipelets" by Jahn and Henkel (2013). It describes pipeline stages which are self-organizing, by monitoring computational demands and communication time, and dynamically remapping tasks using the resulting information. As such, pipelets are able to group tasks such that the communication overhead is minimized and bottlenecks are reduced.

Since the pipeline pattern is very well applicable to the concurrency tools built into the Go programming language, a short example implementation of the pattern in Go will be presented in the next section.

5. The "Go" programming language

This section will give a short overview of the programming language "Go", as well as provide a comparison to C, since it is the most common programming language used on embedded platforms (Douglass, 2011). To wrap up this section, an example implementation of the pipeline pattern will be presented and explained.

Go is a general-purpose programming language first released in 2009, and currently being developed as an open-source project (Google, 2013). Its main goal is to increase developer productivity, especially when working in environments featuring multicore processors, networked systems and massive computation clusters (as such, it tries to solve MIMD problems). It also aspires to be easy to learn. An additional focus when developing the language was the reduction of compile time, since some modern languages, like C++, suffer from very high compilation times for larger projects (D'Angelo et al., 2012), which in turn reduces developer efficiency.

Pike (2012) states that most of the languages widely in use today were created in environments that are, in large parts, unrelated to the current computing landscape. He is referring to the paradigm shift from sequential to concurrent applications, where it becomes necessary to reason about and implement concurrent software in order to exploit the available resources. Languages such as C/C++, Java and Python were built when the importance of multi-processor programming had not yet developed, and as such, all of the parallelism concepts that can be found in them are the result of the adjustment of the language to requirements that developed after the language was first released. As an example, the building blocks for concurrent applications that can be found in the `java.util.concurrent` package were introduced in Java version 1.5, eight years after the initial language release (Peierls et al., 2005). As a result, writing concurrent applications in these languages can often feel like a workaround (Pike, 2012).

Go in turn was developed with concurrency as one of the main language features. The compiler inserts a small runtime environment into every compiled binary, which, apart from providing convenient features such as garbage collection, is able to distribute the so-called "goroutines" onto a pool of operating system threads. As a result, goroutines are much more "lightweight" than OS threads, since no interaction with the operating system is necessary to create and schedule them (other than the management of the thread pool). Creating a goroutine does not require additional work for the developer compared to calling a function, as can be seen in Listing 2.

```go
func some_function() {
    fmt.Println("Hello world")
}
some_function()      // Normal invocation
go some_function()   // Concurrent invocation
```

Listing 2: Example of the sequential invocation of a function compared to the concurrent execution using a goroutine.

The communication between goroutines is realized through channels (the language keyword being `chan`), which have a capacity defining how many messages each channel can hold. Go's communication model can thus be regarded as a "message-passing" model. If any goroutine tries to insert or retrieve data from a channel, and if this cha is full or empty respectively, then the invoking goroutine will block until data becomes available. As a result, channels are also the synchronization primitive in Go.

```go
// Create a channel with a capacity of one
channel := make(chan int, 2)
channel <- 5          // Write data to a channel
out := <- channel     // Retrieve data
```

Listing 3: Syntactical overview of channel usage in Go.

The language includes additional features, which do not deal with concurrency, but have been added for developer convenience. Some of these features include multiple return parameters, slices (which provide a more powerful interface to C-style arrays), garbage collection (which will be discussed in the comparison to C) and built-in dependency management.

mention defer?

5.1. Comparison to C

This section highlights the most important differences between Go and C, where the importance is measured against the potential impact a difference may have when developing embedded applications.

The most important difference between Go and C is Go's usage of garbage collector. In C, memory must be managed by the programmer, which is an important property for embedded systems, since memory is often a very limited resource there. For the Android operating system, which mostly runs on battery powered embedded platforms (smartphones and tablets), the user-facing software ("apps") is written in Java. Since Java is a garbage collected language, the DalvikVM executing the Java bytecode also features a garbage collector. The fact that such a widely used platform uses garbage collection does not imply that it is a good fit for embedded platforms, though. In a recent article, Crawford (2013) analyzes the impact of garbage collection on mobile platforms, coming to the conclusion that garbage collection performs well as long as the available memory exceeds the needed memory by a factor of 6 or more. "In a memory constrained environment garbage collection performance degrades exponentially". As a result, the impact of Go's garbage collector on application performance will have to be closely evaluated.

Another important difference is the existence of goroutines. In C, in order to create a concurrent execution path, an operating system thread must be created. Go on the other hand creates a fixed pool of operating system threads, onto which created goroutines are mapped dynamically at runtime. This is achieved through Go's own scheduler, which is embedded into every compiled application. Having two separate schedulers (from the operating system and from Go) may have adverse effects on performance, since both schedulers are independent. While it would be possible to assign the highest operating system scheduling priority to the Go process (similar to the approach by Jestel (2013)), it would still be difficult to predict the timing behavior of a Go application. Accordingly, Go cannot be used for hard real-time applications in its current state, and its applicability for a soft real-time environment remains to be determined.

At this time it is unfortunately not yet possible to present proper results of a performance comparison between the two languages. This is due to the fact that benchmarking in general is a "notoriously delicate" task. It becomes even more complicated when one is trying to perform so-called micro-benchmarks, where the performance of two short piece of code (as opposed to two applications) written in different languages is to be compared (Gil et al., 2011). One of the reasons for this complexity is that it is inherently complicated to write two pieces of code in two different languages, such that both code pieces do exactly the same thing in the most performant way possible. Writing such code requires advanced knowledge of the language, as for example choosing the wrong datastructure in one of the code pieces may result in completely misleading benchmark results, from which false conclusions will be drawn.

To summarize it can be said that, while Go is a compiled and statically typed language that has the potential to reach near-C performance, especially the scheduler and memory management model can lead to performance issues when running on embedded hardware. Whether or not Go can be productively used on embedded platforms warrants extensive practical research.

5.2. Example: Pipeline Pattern in Go

In this example, an implementation of a pipeline in Go will be shown. The pipeline pattern was chosen because it is likely that it will be used for the parallelization of the obstacle detection system. While the pipeline pattern is especially suited for implementation in Go, other popular patterns are also idiomatically implementable in Go, and are usually shorter or at least have the same length then an equivalent implementation in Java or C++ (Schmager et al., 2010). For the sake of brevity, the operations which the pipeline performs are very simple: it first multiplies every input by two (Stage #1) and then prints the value to the console (Stage #2). The implementation consists of two parts: the initializer (Listing 4) and the stages (Listing 5).

```
 1 func main() {
 2     N := 100 // Number of elements to process
 3 
 4     stage1_in := make(chan int)
 5     stage1_out := make(chan int)
 6     stage2_out := make(chan int)
 7 
 8     go func() {
 9         // Send N integers to the
10         // stage1_in channel
11     }()
12 
13     go stage1(stage1_in, stage1_out)
14     go stage2(stage1_out, stage2_out)
15 
16     // Wait for all elements to be processed
17     for i := 0; i < N; i++ {
18         <-stage2_out
19     }
20 }
```

Listing 4: The pipeline initializer, which creates the necessary channels, fills the pipeline with some elements and invokes the stages.

```
 1 func stage1(in chan int, out chan int) {
 2     for {
 3         current := <-in
 4         op := current * 2
 5         out <- op
 6     }
 7 }
 8 
 9 func stage2(in chan int, out chan int) {
10     for {
11         current := <-in
12         fmt.Println("Value: ", current)
13         out <- current
14     }
15 }
```

Listing 5: The first pipeline stage multiplies every element by two, the second pipeline stage prints the elements to the console.

The initializer first creates the necessary channels for the pipeline, by invoking `make()` with the type "`chan int`" (lines 4–6). The pipeline only needs three channels, since the first stage's output channel is also the second stage's input channel. The initializer then popula[te]he first stage's input channel with values (ommited in Listing 4), and invokes both pipeline stage functions as goroutines (line 13+14), so that they execute concurrenly. Lastly, the initializer waits until all values have been processed by the second stage (line 17–19).

The stages are a perfect match for the pipeline stage structure previously described in subsection 4.2. When looking at the first stage, the first line is the function signature, taking the input and output channel as paramters. In Go, a for-loop such as "`for { }`", that is without parameters, executes an infinite loop. As such, the pipeline stages reads data from the input channel into the "`current`" variable until the program terminates (line 3). Since reading from an empty channel is a blocking operations, the loop does not perform busy-waiting, which would waste many CPU cycles. On line 4 the operation on the current element is performed, and on line 5 the processed data is written to the output channel of the stage.

Importantly, the shown pipeline would not have any performance benefit, since the time it takes to communicate between the channels is much larger than the operations within the stages (the multiplication and the writing to the standard output). However, it would be very straight forward to replace the simply operations shown in the example by more complex operations .

This still sounds a bit incomplete

6. Summary

Komprimierte Darstellung der wesentlichen Inhalte

Ausblick (wie geht es weiter?)

Schwarz: Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications (Expert Guide)? http://goo.gl/vM360

Hier im Ausblick erwähnen, dass Performance-Vergleiche anhand eigener Anwendungen folgen, bezogen auf SIMD stuff aber auch Go auf embedded (vs. C/C++)

Literatur Recherche zu Concurrency und aktuellen MPSOC. (Multiprocessor System-on-Chip: Hardware Design and Tool Integration) Paralleliserung von Laserscanner und Kamera Datenstromverarbeitung unter Linux mit GO im Wettstreit mit C-Threads-pur.

Mention that (Herlihy and Shavit, 2012) will be used as an additional "fundation work", since it was revised in 2012 and thus is newer, but was not available to the author until now.

Ausblick: As a result, the impact of Go's garbage collector on application performance will have to be closely evaluated (also scheduling stuff)

Mention that the NEON Alchemy Lab stuff must be confirmed.

Schwarz: The Art of Multiprocessor Programming sollten wir auch auf dem Zettel haben, da hier eine neue Auflage vorliegt. http://goo.gl/TZU73

short essay outlook: In the course of my master seminar, these hypotheses will be evaluated by extracting performance critical elements from the mentioned FAUST application and implementing them in Go and C, using state of the art dataflow parallelization algorithms. These implementations will then be compared to their original (non-parallelized) counterparts, in terms of performance, ease of implementation and implementation complexity.

References

ACM (2013). Digital Library.
URL: *http://dl.acm.org* (last accessed: 2013-07-19)

Andresen, E. (2013). *ARM-basierter MPSoC unter Linux für eine Fahrspurführung mit Bildverarbeitung*, Master thesis, Hamburg University of Applied Sciences.

ARM Ltd. (2013). ARM Compiler Toolchain - Assembler Reference v5.03.
URL: *http://infocenter.arm.com/help/topic/com.arm.doc.dui0489i/DUI0489I_arm_assembler_reference.pdf*

Campeanu, G. (2013). Support for high performance using heterogeneous embedded systems: a Ph.D. research proposal, *Proceedings of the 18th international doctoral symposium on Components and architecture*, WCOP '13, ACM, New York, NY, USA, pp. 19–24.
URL: *http://doi.acm.org/10.1145/2465498.2465502*

Cantrill, B. and Bonwick, J. (2008). Real-World Concurrency, *Queue Magazine - The Concurrency Problem* **6**(5): 16–25.
URL: *http://doi.acm.org/10.1145/1454456.1454462*

Crawford, D. (2013). Why mobile web apps are slow.
URL: *http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/* (last accessed: 2013-07-20)

D'Angelo, G., Ferretti, S. and Marzolla, M. (2012). Time warp on the go, *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, pp. 242–248.
URL: *http://dl.acm.org/citation.cfm?id=2263019.2263057*

Douglass, B. P. (2011). *Design Patterns for Embedded Systems in C*, first edn, Newnes. ISBN: 978-1-85617-707-8.

FAUST Project (2013). FAUST - Fahrerassistenz und autonome Systeme.
URL: *http://faust.informatik.haw-hamburg.de* (last accessed: 2013-05-07)

Finlayson, I., Davis, B., Gavin, P., Uh, G.-R., Whalley, D., Själander, M. and Tyson, G. (2013). Improving processor efficiency by statically pipelining instructions, *SIGPLAN Not.* **48**(5): 33–44.
URL: *http://doi.acm.org/10.1145/2499369.2465559*

Flynn, M. (1972). Some Computer Organizations and Their Effectiveness, *Computers, IEEE Transactions on* **C-21**(9): 948–960.

Gil, J. Y., Lenz, K. and Shimron, Y. (2011). A microbenchmark case study and lessons learned, *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, ACM, New York, NY, USA, pp. 297–308.
URL: *http://doi.acm.org/10.1145/2095050.2095100*

Google (2013). The Go Programming Language.
URL: *http://golang.org/* (last accessed: 2013-07-18)

Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming*, revised edn, Morgan Kaufmann. ISBN: 978-0123973375.

IEEE Xplore (2013). Digital Library.
URL: *http://ieeexplore.ieee.org* (last accessed: 2013-07-19)

Jahn, J. and Henkel, J. (2013). Pipelets: self-organizing software pipelines for many-core architectures, *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, EDA Consortium, San Jose, CA, USA, pp. 1516–1521.
URL: *http://dl.acm.org/citation.cfm?id=2485288.2485647*

Jang, M., Kim, K. and Kim, K. (2011). The performance analysis of ARM NEON technology for mobile platforms, *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, ACM, New York, NY, USA, pp. 104–106.
URL: *http://doi.acm.org/10.1145/2103380.2103401*

Jestel, A. (2013). *Software-Partitionierung einer Laserscanner- basierten Objekterkennung auf einer MPSoC-Plattform mit Android*, Master thesis, Hamburg University of Applied Sciences.

Johannsen, B. (2013). *Hardware/Software-Codesign für ein SoC-basiertes Spurführungs-system*, Master thesis, Hamburg University of Applied Sciences.

Mattson, T., Sanders, B. and Massingill, B. (2004). *Patterns for parallel programming*, first edn, Addison-Wesley Professional. ISBN: 0321228111.

NEON Alchemy Lab (2013). NEON Tutorial Part 1.
URL: *http://armneon.blogspot.de/2013/07/neon-tutorial-part-1-simple-function_13.html* (last accessed: 2013-07-18)

Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D. and Holmes, D. (2005). *Java Concurrency in Practice*, Addison-Wesley Professional. ISBN: 0321349601.

Pike, R. (2012). Go at Google, *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12* p. 5.
URL: *http://dl.acm.org/citation.cfm?doid=2384716.2384720*

Preud'Homme, T., Sopena, J., Thomas, G. and Folliot, B. (2012). An Improvement of OpenMP Pipeline Parallelism with the BatchQueue Algorithm, *2012 IEEE 18th International Conference on Parallel and Distributed Systems* (i): 348–355.
URL: *http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6413677*

Schmager, F., Cameron, N. and Noble, J. (2010). GoHotDraw: evaluating the Go programming language with design patterns, *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, ACM, New York, NY, USA, pp. 10:1—-10:6.
URL: *http://doi.acm.org/10.1145/1937117.1937127*

Wang, C., Chandrasekaran, S., Sun, P., Chapman, B. and Holt, J. (2013). Portable mapping of openMP to multicore embedded systems using MCA APIs, *SIGPLAN Not.* **48**(5): 153–162.
URL: *http://doi.acm.org/10.1145/2499369.2465569*

Xilinx (2013). Zynq-7000 All Programmable SoC Overview.
URL: *http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf*

Zang, W. and Gordon-Ross, A. (2013). A survey on cache tuning from a power/energy perspective, *ACM Comput. Surv.* **45**(3): 32:1—-32:49.
URL: *http://doi.acm.org/10.1145/2480741.2480749*