technische universität
dortmund

Master thesis

An efficient real-time capable multi-core module framework for the humanoid robot NAO

Aaron Larisch
aaron.larisch@tu-dortmund.de

July 13, 2020

Reviewers:
Prof. Dr. Uwe Schwiegelshohn
Prof. Dr. Jian-Jia Chen
Supervisor:
Dipl.-Inf. Ingmar Schwarz

Institut für Roboterforschung
Abteilung Informationstechnik

Fakultät für Informatik
Lehrstuhl für Eingebettete Systeme

Contents

Chapter 1

Introduction

Robots play an increasingly important role both in industry and in everyday life and become an important part of it. Autonomous robots in particular have to process a large amount of data provided by the sensors that allow them to perceive their environment. This processing is a big challenge since the available resources of a mobile robot are limited and must therefore be used in the most efficient way. In recent years, a trend towards multi-core processors has developed on the hardware side that also require changes on the software side to be used properly. This thesis follows this trend and improves the multi-core usage of an existing robot framework for the humanoid robot NAO that is used by the team Nao Devils of the Technical University of Dortmund during the RoboCup competition.

This chapter first presents details about the RoboCup competition and the NAO robot and then explains the motivation and goal of this thesis.

1.1 About RoboCup

RoboCup is a worldwide competition where teams of universities from different countries and continents compete in different challenges in the field of autonomous robots and artificial intelligence. Playing soccer as a *grand challenge problem* is the main focus in most RoboCup leagues [22] and acts as an application of current research and development in this area. Nevertheless, there are also leagues that develop robots for rescue, logistics, or home assistance. All participants come together every summer to perform a tournament that honors the best teams in each discipline. The tournament-style competition allows the teams to compare their work with others and shows the performance in real-world scenarios.

RoboCup's objective is to support the research on autonomous robots by defining an ultimate goal for the future: to beat the current World Cup soccer champion in 2050 with humanoid robots according to the official FIFA rules [23]. This way, RoboCup

wants to accelerate the development of the required hardware and software under the assumption that the new gained knowledge can be reused for other purposes as well. For example, similar image processing algorithms may be used for autonomous cars, similar robots may be used for nursing in the future or artificial intelligence may guide a robot to rescue a buried human out of a collapsed building. After the competition, RoboCup also organizes a symposium that allows the teams to present their research in form of posters or presentations and to get in contact with other people to discuss their work. Furthermore, all successful teams have to publish at least a part of their work, for example, the program code, to qualify for the RoboCup the following year.

The robot soccer team of the Technical University of Dortmund called *Nao Devils* has been participating in the *Standard Platform League (SPL)* since 2008. The SPL forces all teams to use the same type of robot. This robot called NAO is manufactured by SoftBank Robotics. It allows the teams to focus on the software development and forces them to exploit the given hardware as best as possible. During a game, five NAOs per team fight for the ball on a field measuring 9 by 6 meters and try to score as many goals as possible within 20 minutes of playing time [24]. Recently, the team Nao Devils participated in RoboCup in Sydney 2019 and took 4th place out of 20.

The team consists of employees, students, and volunteers that continuously improve the huge code base and implement new features, enhance the image processing, develop better motion algorithms, or adjust the behavior to deal with new situations and rule changes. For preparation for RoboCup in summer, the team also participates regularly in the German Open that takes place every year in spring. The German Open is a smaller version of the RoboCup for local teams from Germany and surrounding countries and is held in Magdeburg.

1.2 About the NAO

The NAO is a battery-powered humanoid robot manufactured by SoftBank Robotics, formerly known as Aldebaran Robotics. It has an integrated x86 processor, two HD cameras, 25 servo motor controlled joints, an inertial sensor unit, speakers, microphones, offers built-in WLAN, LAN, and much more. SoftBank Robotics releases new versions of the NAO from time to time that fix issues, improve the product, and introduce new features. In figure 1.1, the new NAO version 6 is shown, which has gotten a newer CPU, better cameras, some motor upgrades, and a few design and software changes. Instead of just one single-core CPU with simultaneous multithreading, the new version is equipped with a quad-core Intel Atom E3845 CPU from the Silvermont series, which runs at 1.91 GHz and also features a small integrated General Purpose GPU. It is supported by 32 KB of L1 instruction cache and 24 KB of L1 data cache per core and it has a shared 1 MB L2 per two cores [8]. Furthermore, the amount of available main memory increased from 1

Figure 1.1: Three NAOs version 6 by Nao Devils defending the soccer goal in a game against Nao-Team HTWK Leipzig during Robocup 2019 in Sydney.

GB to 4 GB [28]. As the previous version, each NAO runs a Linux-based operating system with the PREEMPT_RT patch [13], which ensures the required timings for controlling servo motors and processing sensor data.

1.3 Motivation

While the rules are changing every year aiming to get as close as possible to the official FIFA rules, the demands on software and hardware are increasing as well. The latest rule-changes include natural lighting conditions, a new black-and-white spotted ball, which is harder to distinguish from other objects, more complex behavior requirements like kick-in or penalty kick, and a new carpet with artificial turf hampering the walk [24]. Over the last years, the Nao Devils team met the increasing requirements using more demanding algorithms. For example, they have recently designed

- a platform to execute and train CNNs using the YOLO (You Only Look Once) framework for image segmentation to detect other robots or to generate ball hypotheses [20, 29],
- an implementation of a multi-hypotheses Kalman filter for world modeling [10],
- a walking engine based on the Flexible Linear Inverted Pendulum Model [30],
- a path planning algorithm with obstacle avoidance,
- a heuristic whistle detection using FFT analysis

and much more, which is explained in detail in the team report [27]. The robot executes most of these algorithms simultaneously and in real-time on the internal CPU, which is

highly limited, especially in the previous NAO version 5. Since the old NAO version only offered a single-core CPU, the current framework does not include functionality to distribute the work onto additional cores. When running the existing program code on the CPU of the new NAO version 6, it is still limited by the speed of a single core, which increased only slightly compared to the old version. As a result, the remaining three cores of the new quad-core CPU are mostly unused.

This thesis develops an approach that overcomes this limitation by designing an architecture that distributes the workload over multiple cores while maintaining the modular structure of the framework and keeping changes to individual modules at a minimum. The objective is to reduce the overall execution time in order to free resources for additional more demanding algorithms. The added parallelization should be as transparent as possible to developers working on a module and the framework should abstract from the technical details. Furthermore, the framework must remain efficient and real-time capable while being easy to use. Therefore, the debugging functionalities must not be restricted, but on the contrary, new possibilities should be created to make the added parallelization easy to understand for other developers.

1.4 Structure

The thesis consists of five main parts. First, we give an overview of the terms framework and real-time, which are used as a basis in the following chapters. Second, we analyze the current situation. Therefore, we outline the current design of the framework, formulate the requirements on the parallelization and reveal possible improvements offered by the framework. Third, we cover existing parallelization approaches in other robotic frameworks as well as libraries that already handle the parallelization of program code and we consider if they may be suitable to integrate. Fourth, we explain the implementation details of the best promising solution as well as the challenges, which occurred. Fifth, we evaluate the solution during several small test games. Therefore, we investigate the robot's performance executing the adjusted framework in different scenarios and determine the gained speed up. Finally, we discuss if the objective was achieved and give possible starting points for further improvements.

Chapter 2

Prerequisites

This chapter provides some basic information concerning frameworks and real-time used in the following chapters.

2.1 Framework

In software development, a framework consists of multiple classes, which first increase the reusability of program code in a software project and second preserve an abstract design the programmer should follow in order to interact with it. This helps structuring the program, reduces the need for individual design choices the programmer has to take, and provides reusable components, which might be helpful to reduce complexity [11]. In this case, the Nao Devils framework introduces the concept of modules and representations, cares about the class instantiation and execution, manages the memory, and offers interfaces for communication between different modules, threads, or computers. Furthermore, it is integrated into a simulation software, which offers a 3D simulation, has integrated debug interfaces, and much more. Details on that will be explained in chapters 3 and 5.

2.2 Real-time

When talking about the correctness of an application or algorithm, it is often referred to the logical correctness. However, in several circumstances, this is not enough because temporal constraints come into play and have to be considered. This the case in time critical applications whose environment forces the algorithm to react to an input within a given time. For example, a program that controls a physical experiment using some actuators and sensors must keep the system within its limits to prevent damage. If the reaction to an external event takes too long, the experiment may fail.

The term real-time means that a task must finish its execution within a given time span called deadline. If the task finishes too late, its result might be less usable, completely

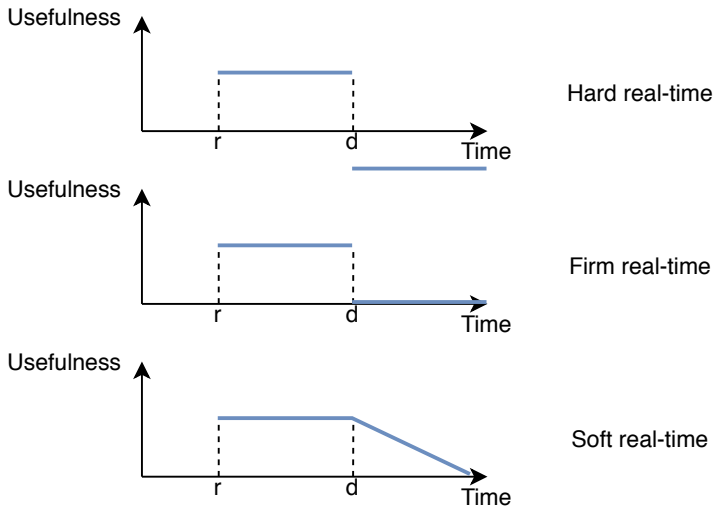


Figure 2.1: Usefulness of the task's result after missing the deadline d.

unusable, or it might cause harm, injury, or damage. When considering the consequences of a missed deadline, tasks can be grouped into three categories, which are also visualized as graphs in figure 2.1 [3]:

- A *hard* real-time task missing its deadline may cause a catastrophe. For example, an autonomous driving system in a car must react to sensor inputs on time under any circumstances to prevent collisions with appearing obstacles. If the deadline is missed, the usefulness of the result is negative.
- A *firm* real-time task missing its deadline makes the generated results completely useless. For example, when playing a video or audio stream, frames that are decoded too late may be skipped and result in a reduced playback quality but do not cause harm. If the deadline is missed, the usefulness is zero.
- A *soft* real-time task missing its deadline makes the generated results less useful. For example, in human-machine interaction, a delayed reaction to mouse movement or keyboard input might annoy the users but do not bother them if it happens rarely. If a deadline is missed, the usefulness decreases over time.

The software on the NAO can be classified into three categories:

- Motion control has firm real-time requirements. To ensure smooth movements of the robot, it has to control its servo motors constantly based on gained sensor measurements. The microcontrollers for the servo motors require periodic position updates from the CPU to accomplish that. If the program misses the deadline and updates the target position too late, the robot might stutter. Nevertheless, if this happens rarely, the robot should be able to continue its motion and the missed deadline does not result in a complete system failure.

- Cognition and behavior have soft real-time requirements. The robot has to avoid obstacles and makes its decisions based on the images captured by the cameras during the game. The processing of each image must be completed until the next one is available to keep up with the constant frame rate and to react to changing conditions quickly. Although a delay in processing increases the response time and permanently results in poorer game performance, it does not affect the robot's stability and does not make it fall down.
- Program code for debugging, logging, and network communication is non-time-critical and does not affect the autonomy of the robot itself.

Chapter 3

Analysis

In this chapter, we illustrate the current framework design and provide possible starting points that are suitable for parallelization. Furthermore, we discuss these staring points and outline their advantages and disadvantages. Finally, we define a list of requirements for the implementation that ensure a correct and efficient operation of the robot framework.

3.1 Current Situation

The current Nao Devils framework is based on B-Human's code release from 2015 [26]. B-Human is the SPL team of the university of Bremen, which releases the majority of their code to the public after every RoboCup and offers a good starting point for many other teams. The Nao Devils team has adjusted many parts to fulfill their needs and replaced the majority of modules by their own ones. The last state can be found in the code release from 2019 [17].

The framework is entirely written in C++ and consists of the following three main parts:

1. A flexible robot infrastructure,
2. a 3D simulator with debug interface, and
3. several deploy and setup tools for cross-compilation and installation procedures.

Infrastructure The robot infrastructure follows a *blackboard architecture* [9] that consists of *representations* which are provided and required by *modules*. A representation is a C++ struct that contains arbitrary data and is stored in a global table "blackboard". A module is a C++ class that operates on these representations. Every representation is provided by exactly one module and multiple other modules can require it and gain read-only access to it. This allows different modules to communicate over these representations and to separate program code and knowledge into logical units and small pieces. The design

principle is to keep the modules as small as possible to reduce the complexity and to allow easy debugging by having many well-defined interfaces. During the framework start up, the module manager reads a configuration file that maps each representation to a module that becomes the active provider for it. Moreover, it ensures the correct execution order of every module and keeps track of the dependencies. Furthermore, the module manager can be reconfigured during runtime via commands. That allows to compare, for example, two image processing algorithms, which are implemented in different modules, by interactively switching between both of them. At any time, each representation is provided by exactly one module. Each time the configuration changes, the module manager recalculates the dependencies and the framework continues the execution.

```
STREAMABLE(BallPercept,
{
  ENUM(Status,
  {,
    notSeen,
    seen,
  }),

  (Status)(notSeen) status,
  (Vector2f)(Vector2f::Zero()) positionInImage,
  (float)(1.f) radiusInImage,
});
```

Listing 3.1: BallPercept representation.

A representation is a data type that is generated via the C++ macro STREAMABLE. The example representation BallPercept can be seen in Listing 3.1. In this context, streamable means that all data can be serialized into either a human-readable string, which can be printed out, or a binary data stream, which can be transferred over the network or saved on disk. Therefore, the macro generates a C++ struct that contains the specified properties with their given default values and a serialization function. This serialization function is able to convert all default data types into the required binary or textual format but can be implemented manually if necessary, for example, when using complex data types.

New modules are defined similarly. An example definition of the IMUModelProvider is shown in Listing 3.2. MODULE(...) is another C++ macro that generates an abstract class, which the programmer must implement, and uses the following keywords:

- PROVIDES(<REPRESENTATION>) adds an update method that gets called once per cycle by the module manager passing a reference to the given representation inside the blackboard. This function is pure-virtual and must be implemented by

```
MODULE(IMUModelProvider,
{ ,
  REQUIRES(InertialSensorData),
  REQUIRES(FrameInfo),
  USES(MotionInfo),
  PROVIDES(IMUModel),
  LOADS_PARAMETERS(
  {,
    (bool)(true) enableWhileWalking,
    (IMUModelAccelerationFilterParameters) standAcceleration,
    (IMUModelAccelerationFilterParameters) walkAcceleration,
    (IMUModelRotationFilterParameters) standRotation,
    (IMUModelRotationFilterParameters) walkRotation,
    (float)(9.81f) gravity,
  }),
});
```

Listing 3.2: IMUModelProvider module definition.

the sub-class. It performs all the necessary computations to fill the passed representation.

- REQUIRES(<REPRESENTATION>) adds a class property that gives read-only access to the specified representation inside the blackboard. Furthermore, it ensures that the representation is filled by the configured provider before executing any update methods of the module being declared.

- USES(<REPRESENTATION>) is similar to REQUIRES, but it does not guarantee that the representation is filled before the execution of the module and the data may be generated in the cycle before. This is needed to resolve cyclic dependencies and should be used only when necessary.

- LOADS_PARAMETERS([<PARAMETERS>]) adds class properties as configuration parameters and is related to the STREAMABLE macro explained earlier. The framework loads all specified parameters automatically from a configuration file, which can be edited easily via the user interface explained later.

For each module, the framework keeps track of the required and provided representations. Before the execution of the first cycle, the module manager brings the providers for all representations into the correct order to ensure that all requirements for each module are met on execution. If the requirements cannot be fulfilled, for example, because of a missing provider for a required representation or circular dependencies, the module manager prints out an error message and aborts the start-up.

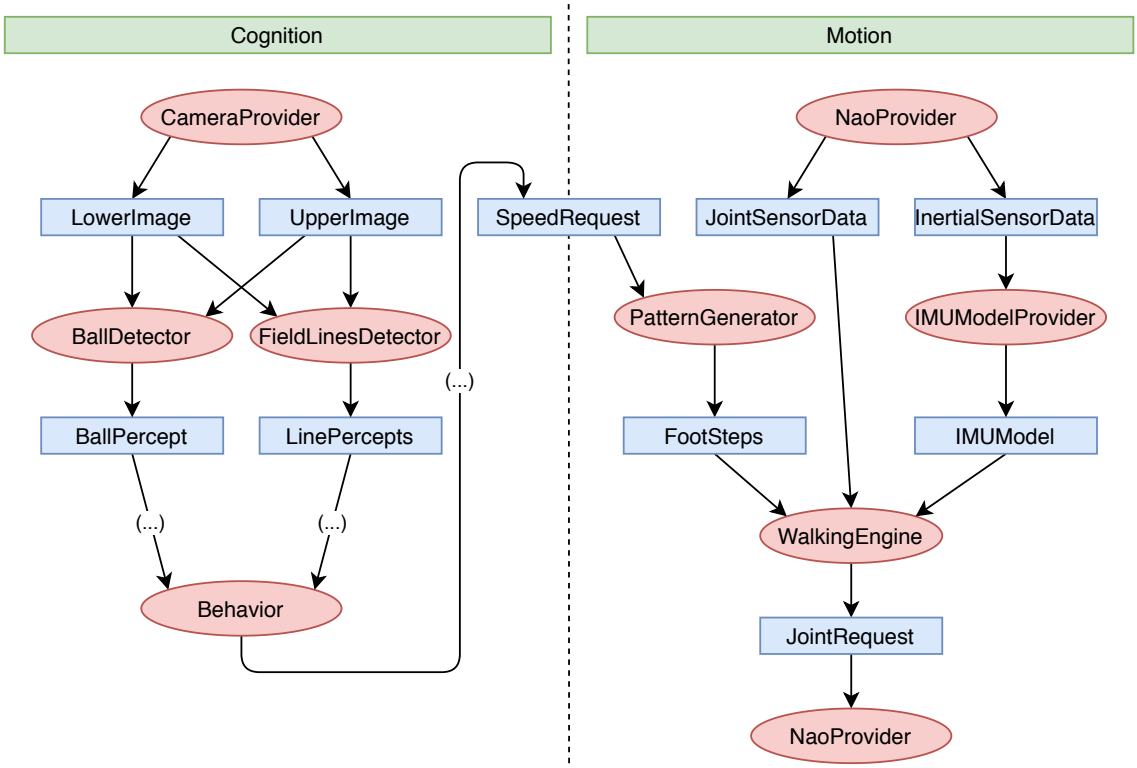


Figure 3.1: Simplified directed acyclic graph modeling the dependencies between modules shown in red and representations shown in blue.

As an example, figure 3.1 visualizes these dependencies for some representations and modules currently present in the framework. Modules are shown in red, representations are shown in blue. Any module that provides representations that are required by a subsequent module must be executed first. The execution of the BallDetector cannot be started before the execution of the CameraProvider, which provides the LowerImage and UpperImage. Analogous to this, the execution of the Behavior must wait until its requirements are also met. Furthermore, all modules are divided into two parts motion and cognition shown in green, which are executed in two separate threads. Both threads operate mostly on separate memory regions to avoid race conditions on the data. That also means that there are currently two blackboards containing the representations for motion and cognition and there are also two graphs containing the dependencies between the modules and representations.

There are several reasons for this separation:

1. Different triggers: a new motion cycle is triggered by new sensor data being available from the robot via the NaoProvider and it is finished by sending new actuator data to the server motors. In contrast to that, a new cognition cycle is triggered by new images being available from the CameraProvider and finished by sending the required walking speeds to the motion cycle.

2. Different frequencies: new sensor data is available 83 times per second and new image data is available 30 times per second. Furthermore, the actuator data should also be sent 83 times per second to the servo motors to ensure smooth and stable motions.

3. Different priorities: in order to ensure the timings of both threads, they are assigned a Linux real-time priority. However, to further ensure the firm real-time requirements explained in section 2.2, the motion cycle is preferred over the soft real-time cognition cycle by giving it a higher priority. This way, the motion cycle is able to preempt the execution of the cognition cycle in case new data is available that has to be processed first.

Data that is shared between both threads is copied from one blackboard to the other at the end of each cycle. In the shown example in figure 3.1, the representation SpeedRequest is shared and acts as a connector between both threads. Additionally, a message queue is used as a data buffer to ensure lock-free operation and independent data access. This way, neither the sending nor the receiving thread has to wait for the other to finish its operation and all data is consistent and up-to-date. Because of these two copy operations, the amount of shared data should be kept as low as possible and is only performed for small representations where necessary.

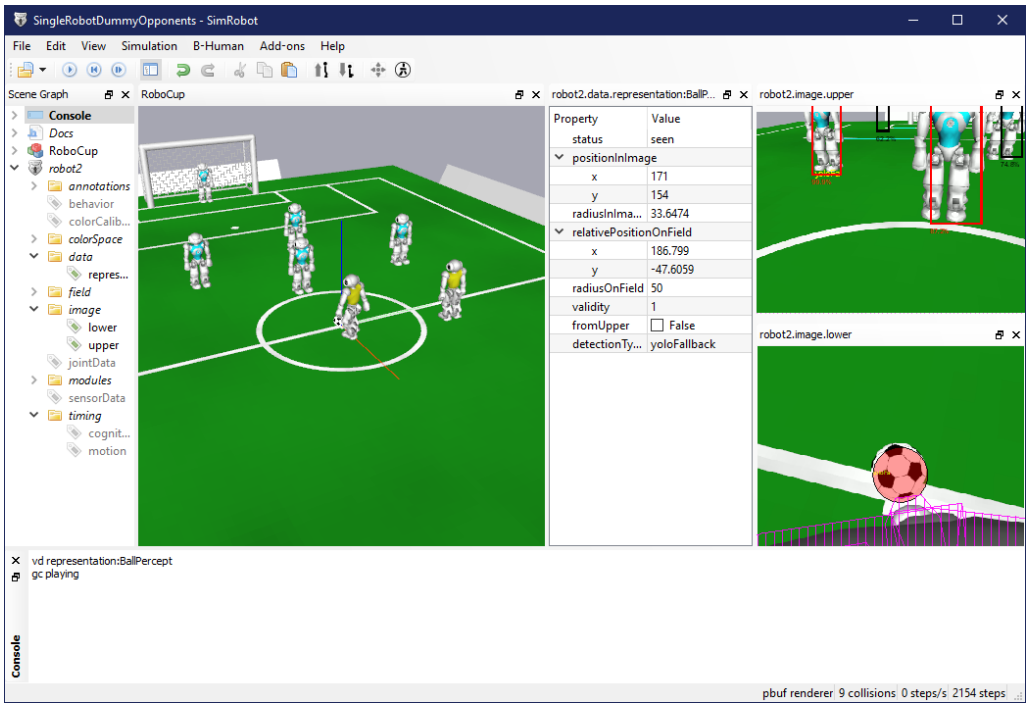


Figure 3.2: Simulator screenshot.

Debugging The framework also offers a simulator shown in figure 3.2, which has three main functionalities:

- Simulating one or multiple robots in a 3D scene with generated physics,
- connecting to a physical robot via network, or
- replaying a recorded log file, for example, out of a real game.

The 3D simulation is based on the Open Dynamics Engine (ODE) and OpenGL. During the simulation, modules that usually implement the connection to the physical robot are replaced by a module that is provided by the simulator. This way, all representations containing real sensor data are now filled with the calculated values and the remaining module infrastructure runs in the same way like it would on the physical robot. Similarly, all representations containing the actuator output are transferred to the 3D scene and physics simulation.

Because most important data structures like representations and configurations are streamable, the simulator is able to show a live view of all values that might be interesting. This is also possible while being connected to a physical robot via network, which allows real-time debugging of errors that might not happen during simulation. Additionally, the developer can set each representation to a fixed value manually. That allows, for example, to override the SpeedRequest and set the robot to a fixed position. This way, the developer can test the image processing and put different objects in front of the cameras, without bothering about a walking robot. This mechanism also allows to test a module with user-defined input values if the developer overwrites the required representations of a module with test data and monitors the module's output.

In order to examine errors that, for example, happened during a game, the robot can save configured representations into a file automatically at the end of each cycle. The generated log file allows to replay the scenario in the simulator for debugging purposes and also allows to test new image processing algorithms in old recorded games.

Compilation The framework's compilation process is handled by a tool called Mare[1]. Mare is a cross-platform build automation tool that keeps track of all source files and libraries and is able to generate project files for a variety of IDEs like Microsoft Visual Studio[2], CodeLite[3], and others. Furthermore, it allows to specify include paths and linker options for custom libraries via an own configuration language. That makes it possible to add new C++ libraries easily while taking care of most platform, architecture, and compiler specific adjustments.

[1] `https://github.com/craflin/mare`
[2] `https://visualstudio.microsoft.com/`
[3] `https://codelite.org/`

3.2 Possible improvements

As already mentioned in section 3.1, the current Nao Devils framework does all computations in two threads: motion and cognition. However, the vast majority of CPU resources is especially needed in the cognition cycle for image processing. Because many motion related calculations have been linearized in the framework and the demands on motion have not increased as much as on cognition, the execution time needed for motion is only a fraction of that of cognition. Since the NAO offers a quad-core CPU, but a single thread cannot use more than a single core, only about one quarter of the available computing power is used.

To improve this situation, the workload must be distributed over several threads that must be created. For the current framework, this is possible on three different layers:

1. *Cycle-Layer*: some modules that are currently executed sequentially inside the cognition cycle are moved into new independent ones. For example, the behavior modules can be separated from the image processing and the image processing can be split up into separate cycles for the upper and lower camera. Even the detection of individual features like the ball, other robots, or the field lines can be separated into their own cycles.

2. *Module-Layer*: since all modules build up a graph structure via their provided and required representations, independent modules can be executed simultaneously. As long as all necessary requirements of a module have already been provided, it is ready to run. That way, for example, image processing modules requiring the image data can start in parallel once the camera captures a new frame.

3. *Execution-Layer*: CPU-intensive modules, for example, for the inference of neural networks for the ball recognition, spawn additional threads in order to parallelize the workload. This way, individual modules can process the images for the upper and lower camera simultaneously or evaluate possible ball hypothesis in parallel.

Subsequent, we consider the advantages and disadvantages of each approach:

Adding additional cycles that split up the cognition processing is a quite easy approach from the implementation perspective because the required infrastructure for the execution of two cycles is already available, which can be extended to multiple ones easily. These changes include the addition of more message queues for the communication between them and the choice of a suitable trigger that starts the module execution once all required input data is available. However, on the one hand, this approach has the drawback that more data must be copied because every thread has its own blackboard, which contains copies of the representations provided in other threads. On the other hand, the overall latency, for example, to process a new camera image and generate a new SpeedRequest increases.

If consecutive cycles have long execution times, the first cycle is already triggered again while the previous processing has not yet been completed. Regardless of this, the Nao Devils team considers adding an additional audio cycle for whistle and voice detection in future because new audio data arrives independent at a different frequency than image or sensor data and audio processing will play an increasingly important role in the future of the Standard Platform League in RoboCup.

The parallel execution of independent modules, which are currently arranged in a single logical cycle, requires major adjustments to the module manager. The current implementation uses a precalculated and ordered list of providers that is generated during framework start up and is used for a consecutive execution of each module. Furthermore, the module manager does not currently spawn any additional threads and is executed synchronously in the motion or cognition thread. Whenever a module finishes, the module manager takes over and executes the subsequent module in its list. A parallel execution of modules requires the management of multiple threads while taking care of a sensible module-thread association and the compliance with their dependencies. Nevertheless, this approach is mostly transparent to the module's programmers. In particular, compared to the approach on the cycle-layer explained before, it does not require the manual association of a module into a specific cycle that requires knowledge about all available cycles and their purposes as well as taking care of dependencies that must be considered to minimize copy operations.

The profit of manual parallelization of algorithms inside a single module highly depends on the used algorithms. While processing the upper and lower image in parallel is quite easy in many cases, further division into independent work units is more challenging. Moreover, the current framework does not offer any functionality to spawn multiple threads for a module or to manage a pool of threads. Without a common framework-wide mechanism providing such a functionality, the result would be an increasing number of modules implementing their own thread spawning and joining functionalities and handling the inter-thread communication on their own. That leads to overcommitment of more working threads than available CPU cores, which can reduce the overall performance drastically because of unnecessary context switches.

In summary, to avoid redundant individual thread management inside modules and to keep the configuration effort and complexity for module developers at a minimum, an intelligent framework-wide solution seems reasonable. That excludes approaches that are based on the cycle layer and that require manual adjustments and knowledge needed for optimal performance. Therefore, a dynamic approach considering the inherent parallel graph structure at the module-layer is more suitable while possibly offering an interface at the execution-layer for modules that can be splitted into independent work units and allow additional parallelization.

3.3 Requirements

The goal of this thesis is to execute as many modules as possible simultaneously that are able to run in parallel according to their dependencies. In order to evaluate the different concepts that we introduce in the following chapters, some requirements were defined that need to be considered:

1. The concept utilizes all available CPU cores.
2. The concept prioritizes the motion cycle over the cognition cycle to ensure smooth movements of the NAO.
3. The concept switches efficiently between modules. There are many modules doing very simple and fast calculations that must not be slowed down, for example, because of frequent process switches.
4. The concept requires a minimum of communication. Each thread operates on its own data and minimizes communication overhead for serialization or synchronization.
5. The concept requires no manual adjustments and configurations. Developers working on individual modules should not care about the parallelization happening in the background. At best, they have more available computing time without the need to change their code.

Chapter 4

Framework and library research

After the presentation of the current framework design, possible improvements, and the requirements needed for parallelization, this chapter evaluates existing approaches for parallel programming that are suitable for this purpose. The first section covers multi-core approaches of existing robotic frameworks that are used in the context of Robocup and the second section covers libraries for parallel programming that can be integrated into existing software.

4.1 Approaches for robotics

In the SPL league, many teams use the B-Human framework as a basis for their development. In B-Human's recent code release from 2019, they have improved their multi-core support for the NAO version 6 to increase the resource utilization. Another framework is called ROS, which is used by many robotic teams in RoboCup and industry and uses multiple cores natively by design. We outline their concepts hereafter and analyze if a similar design is applicable for the Nao Devils framework.

4.1.1 B-Human framework

The B-Human framework is also based on the blackboard design pattern explained in section 3.1 and the structure is still quite similar to that used by Nao Devils. In order to improve the utilization of the NAO version 6 CPU, B-Human increased the number of cycles and thus threads and made the assigned modules configurable [25]. In the Nao Devils framework, each module is assigned to a fixed thread at compile time. Instead, B-Human extended the module manager configuration to specify which provider of which representation runs in which thread. This mostly corresponds to the optimizations at the cycle-layer mentioned in section 3.2. This way, B-Human splitted the cognition cycle into three parts: image processing for the upper camera, image processing for the lower camera, and behavior computation using the results generated earlier. That allows to process both

images in parallel. Under the assumption that the motion and behavior calculation usually does not need much time, this still has the drawback that there are not more than two threads doing most of the computations. However, the configuration based approach has the advantage that modules can be easily assigned to another cycle without the need of a recompilation like in the current Nao Devils framework. That allows to experiment with different module to thread configurations and different thread counts easily in order to find the best solution.

Unfortunately, some requirements mentioned in section 3.3 are not met using this approach:

- Requirement 1 because the effectiveness of the parallelization is highly dependent on the configuration and the number of cycles.
- Requirement 4 because communication times increase if large representations are shared and copied.
- Requirement 5 because every developer that adds a new module must decide which cycle it belongs to.

4.1.2 ROS

ROS stands for Robot Operating System, was developed by the *Stanford University* and *Willlow Garage* for their service robots and evolved to a very flexible open-source robot framework. Although the name suggests it, ROS is not a classic operating system itself. Instead, it functions on top of an existing operating system. ROS adds a communication infrastructure and presets a structure that allows the developer to reuse existing code and to develop interchangeable software modules. Moreover, it is designed for decentralized multi-agent operation, allowing multiple robots and computers to act as a single compute cluster [19].

ROS2 is a major update of ROS that addresses some of the issues present in the version before like the lack of support for real-time, embedded systems, and non-ideal network communication. It further focuses more on real-world scenarios than only research applications. Therefore, breaking API changes especially at the communication layer that makes ROS2 incompatible with ROS were made. However, the basic principles outlined hereafter remained the same [6].

The framework utilizes the concept of nodes, messages, and topics [19]. A *node* is an independent software module that performs computations and allows the logical separation between different tasks. Each node sends and receives messages to communicate with other nodes. A *message* contains arbitrary data that is specified via an interface definition language and allows automatic serialization and deserialization. Each message is sent over the network using a specified topic. The *topic* mechanism follows the publisher-subscriber

pattern allowing each node to publish and/or subscribe to multiple topics it is interested in. In ROS, a separate *master* process ensures and manages the correct connection of multiple nodes using the same topics. Thus, messages, which are passed via one of these topics, are transferred to the correct nodes automatically. ROS2 removes the need for a separate master process by replacing it with the Data Distribution Service (DDS) that performs automatic node discovery and adds support for deadlines and fault-tolerance [14]. In both cases, once a node receives a new message, it processes the received data and generates new messages for other nodes. That enables inherent multi-core support because each node runs as a separate operating system process which allows to run multiple nodes in parallel that, for example, listen to the same topic. Moreover, each node is able to start additional threads for parallelization if necessary.

Although the concept is quite different from the Nao Devils framework, there are several similarities:

- ROS nodes are similar to modules. Both concepts capsule independent software modules that can communicate with each other.
- ROS messages and topics are similar to representations. Both types contain and transfer arbitrary data to other nodes or modules.
- The ROS master process and the DDS in ROS2 are similar to the combination of blackboard and module manager. Both principles care about the transfer of data and ensure the start of the correct next node or module.

Nevertheless, an approach similar to ROS or ROS2 is not feasible for the Nao Devils framework when considering the requirements from section 3.3. First, the design principle to keep the modules as small as possible is not reasonable when running each one in an individual process like ROS nodes. Since the framework contains modules that consist of just a few operations, the overhead for context switching to another process is too high and requirement 3 cannot be met. Second, passing messages over the network introduces even more overhead and latency, which violates requirement 4. Representations can be very small in the range of a few bytes and the network stack with serialization and deserialization might be more complex than the providing module itself.

In summary, a ROS-like concept significantly increases the module switching and communication times and is not applicable for the Nao Devils framework.

4.2 General approaches

Outside of robotics, there are several libraries for parallel programming that execute a set of tasks on a set of CPU cores or threads. Many of them also support dependencies between different tasks. That problem is very similar to the idea of module parallelization

in the Nao Devils framework. Each module can be compared to a task that is able to be executed only if all of its requirements were fulfilled. Therefore, in this chapter, we cover and compare the libraries OpenMP, Intel Threading Building Blocks, and Taskflow that support the parallelization of multiple tasks with dependencies.

4.2.1 OpenMP

The OpenMP API is one of the widest known and oldest interfaces for parallel programming for shared memory multiprocessors that became an industry standard in 1997. The initiative was supported by application and compiler specialists from the industry who wanted to create a vendor independent standard for parallel programming. Before, each machine had its own programming model that was incompatible with other ones while OpenMP finally merged and standardized the different approaches and specified a common interface that allows to execute the same code on different platforms. The OpenMP specification is available for Fortran and C/C++ [4].

While OpenMP versions until 2.0 concentrate on the parallelization of loops using worksharing and synchronization mechanisms, OpenMP 3.0 introduces the concept of tasks that allows the programmer to write multiple independent work units without the need to care about scheduling [1]. This further improves in OpenMP 4.0 by the addition of task dependencies [18] that may qualify OpenMP for the parallel execution of modules in the Nao Devils framework. However, because OpenMP is designed as a compiler extension and heavily relies on compiler directives, it is very inflexible and runtime changes are very limited. For example, in OpenMP, a new task is defined using the `#pragma omp task shared(x) depend(in: x)` directive that statically specifies the dependency on a shared variable x. Since the Nao Devils framework allows the runtime configuration of modules and representations, a static configuration like this is inapplicable.

Furthermore, the realization of OpenMP as a compiler extension has another drawback. While Clang 9, which is used as C++ compiler for all Linux targets including the NAO, supports at least OpenMP 4.5 [5], the Microsoft Visual Compiler, which is used for the simulation under Windows, supports only OpenMP 2.0 [15] and is thus missing task support. Because the simulated robot running under Windows should perform as similar as possible to the real NAO, it is not sensible to just disable the multi-core support for this platform.

In summary, OpenMP is not applicable for the desired purpose.

4.2.2 Intel Threading Building Blocks

In order to address some of the issues with OpenMP, Intel developed Threading Build Blocks (TBB) in 2007. In contrast to OpenMP, the development of TBB focused on a more dynamic task-based approach right at the beginning and is built as a C++ library

instead of a language extension. Furthermore, it uses native C++ language features instead of additional compiler directives. That makes it mostly platform and compiler independent and functions on Windows as well as Linux using GCC, Clang, and MSVC [21].

```
1  using node_t = continue_node<continue_msg>;
2  using msg_t = const continue_msg&;
3
4  tbb::flow::graph g;
5  node_t A(g, [](msg_t){ std::cout << "A\n"; } );
6  node_t B(g, [](msg_t){ std::cout << "B\n"; } );
7  node_t C(g, [](msg_t){ std::cout << "C\n"; } );
8  node_t D(g, [](msg_t){ std::cout << "D\n"; } );
9  make_edge(A, B);
10 make_edge(A, C);
11 make_edge(B, D);
12 make_edge(C, D);
13 A.try_put( continue_msg() );
14 g.wait_for_all();
```

Listing 4.1: Intel TBB dependence graph example.

Besides loop-based parallelization instructions similar to OpenMP, TBB supports the creation of individual tasks and specification of dependencies during runtime. A minimal example is shown in listing 4.1. In this case, four tasks A, B, C, and D are created in lines 5 to 8 and each of them prints out a letter. Further, the inserted graph edges in lines 9 to 12 require task A to be finished before the execution of B and C and both tasks B and C to be finished before the execution of task D. When using a thread pool of at least two threads, the execution of this task graph can be parallelized. After the completion of task A, the two subsequent tasks B and C are executed concurrently while task D waits for the completion of both. The big advantage is that the library cares about the needed thread spawning, synchronization, and task scheduling and the programmer can focus on the implementation itself.

From the implementation side of TBB, the task scheduling is realized using a work-stealing scheduler [12]. A work-stealing scheduler can be used to execute tasks graphs and the performance has already been examined by [2] in 1999. They showed the expected runtime to be limited only by the sequential part of the workload if the number of processors goes to infinity. Each task is represented as a node and the dependencies between them as a directed edge that compose a directed acyclic graph. The scheduler uses a double-ended queue (deque) for each available processor or thread in this case, which contains all tasks that are ready to run and are executed from the bottom to the top. As soon as a task finishes, all subsequent tasks that have fulfilled dependencies are placed at the bottom of

the deque of the current processor and are likely to be executed next. Whenever a deque is empty, the processor steals tasks from the top of another random non-empty deque.

This strategy is cache-friendly when considering the design of modern processors. CPUs are supported by a hierarchy of cache memory that greatly reduces the execution time whenever required data is available there instead of main memory. Further, at least the fastest and smallest caches are usually exclusive for each CPU core. As already mentioned in section 1.2, the CPU of the NAO has 32+24 KB of L1 cache per core. The strategy stealing tasks from the top of another deque but executing from the bottom ensures that CPU caches are used efficiently were possible because data that was generated by a task recently is probably being used again by tasks depending on it. Whenever data is used that has recently been accessed, it is likely to be stored in the fast CPU cache. That is also very helpful for the Nao Devils framework because representations, which are passed along the dependencies, are very likely to be used again after providing and should be available as fast as possible.

Intel TBB and the used scheduling mechanism introduced briefly seem applicable for the use in the Nao Devils framework and could be used from a theoretical point of view to execute the modules using tasks in multiple threads while taking care of the dependencies using graph edges.

4.2.3 Taskflow

Another more recent approach for task-based parallelization is Taskflow. Tsung-Wei Huang developed the library at the University of Illinois in 2019. It is comparable to Intel TBB but provides an even more modern and easier programming model and improves the performance [7].

```
1  tf::Taskflow taskflow;
2
3  auto [A, B, C, D] = taskflow.emplace(
4      []() { std::cout << "A\n"; },
5      []() { std::cout << "B\n"; },
6      []() { std::cout << "C\n"; },
7      []() { std::cout << "D\n"; }
8  );
9
10 A.precede(B, C);
11 D.succeed(B, C);
12 tf::Executor().run(taskflow).wait();
```

Listing 4.2: Taskflow code example.

The same example from listing 4.1 for Intel TBB reduces to the code shown in listing 4.2 for Taskflow. It also uses a similar work-stealing scheduling algorithm but was further improved. The authors compare Taskflow to OpenMP and Intel TBB using several benchmarks. For example, they evaluate the performance impact of Taskflow on the training of a three-layered DNN and show a speed up of 38% compared to OpenMP tasks and 14% compared to Intel TBB using 16 cores. However, their performance evaluation mainly focuses on very heavy workloads on many-core systems with up to 64 cores that may not be applicable to the NAO [7].

Furthermore, their research also focuses on the development experience of the programmer. Therefore, as already mentioned, they try to keep the interface of the library as intuitive as possible. They compare the effort in terms of time, lines of code, and cyclomatic complexity to integrate the library into an existing project with the other libraries and they note a significant advantage when using Taskflow. Apart from the the implementation details, which highly depend on the programmers skills, the library also offers debug interfaces. When debugging parallel code, it can be hard to understand the reason for bad performance or any error that occurs. Therefore, first, Taskflow allows to print out the current task graph that is currently used for execution and second, it is able to observe the exact scheduling and timing of each task that is started on every thread. When considering the Nao Devils framework this is a big advantage because currently, there is no way to determine the exact order and timing of modules during runtime, which could be helpful when investigating performance issues that happen rarely or are unpredictable.

4.3 Consequences for the framework

This chapter first presented the concepts for multi-core usage in the B-Human framework and ROS. It turned out that both approaches do not fulfill the desired requirements for the Nao Devils framework. Second, it showed that there are several libraries available, which are usable for the desired parallelization of the Nao Devils framework. However, the oldest library OpenMP turned out to be inappropriate for this use case due to being too inflexible and the limited support of the Microsoft Visual Compiler. Intel TBB and Taskflow both follow a very similar paradigm that sounds more promising. From a theoretical point of view, both libraries can be used to fulfill at least four out of five requirements from section 3.3:

- Requirement 1 because the number of threads being used can be configured and set to the physical core count utilizing all available resources.
- Requirement 3 because work-stealing schedulers using a lock-free queue implementation turned out to be suitable for tasks with data dependencies and have low overhead.

- Requirement 4 because the communication times are low and all data remains in shared memory while even cache usage is improved by preferring tasks that depend directly on a finished task.
- Requirement 5 because the libraries require no configuration for module developers.

However, in none of the libraries shown, prioritization features are available. Neither can individual tasks be given priority, nor is the library able to preempt running tasks. Preemption is not even possible for a user-space implementation in Linux. Under the assumption that each task runs program code that is out of control of the framework design itself, the only way to suspend the execution is via interrupt. Because interrupts are handled by the operating system, this mechanism is not available for a user-space preemption of tasks. A solution for that is to keep each cycle in different thread pools and to run two instances of the scheduler, where each one can be prioritized like before using the standard Linux thread priorities. This way, in case a higher prioritized cycle is able to run, the operating system can preempt the less important cycle. This also mets requirement 2.

Both libraries also have support for dynamic tasking that allows to spawn even more tasks out of a running task. When providing an interface to the library for module developers, it can also be used to parallelize the program code of individual modules. This approach allows to realize the execution-layer parallelization mentioned in section 3.2 that can further speed up the execution and which can be used optionally by other developers.

Because Taskflow has better performance, a helpful debug interface, and a clean API which could potentially be used directly by module developers, we use Taskflow for the following chapters to implement the module based parallelization.

Chapter 5

Design and implementation

Based on the results from chapter 4, which illustrated different approaches for parallelization, we implemented the most appropriate approach based on Taskflow for the Nao Devils framework. In this chapter, we outline the details on that as well as the resulting challenges.

5.1 Task graph generation

In order to integrate Taskflow properly, we developed an appropriate mapping that transforms the module-representation graph shown in section 3.1 to a task graph. For the old module execution mechanism, it was sufficient to generate a list of providers that is sorted according to the dependencies. This list was calculated only once during framework start up or after the modification of the module graph and afterwards, each listed provider was executed sequentially in every cycle. During the execution itself, the dependencies were not important anymore.

In contrast to that, due to the absence of a known exact execution order when using a work-stealing algorithm, the Taskflow library needs to keep track of the dependencies during runtime. Therefore, it expects a task graph that models the separate execution units properly. In the following, we explain the details of this conversion. For simplicity reasons, it is assumed that each module only provides representations that are enabled in the module manager's configuration and inactive providers for representations are not considered. However in reality, the framework allows to switch between different modules providing the same representation dynamically during runtime. Whenever this happens, the task graph is generated again.

Let M be the set of modules and R be the set of representations. Then, for an arbitrary module $m \in M$, the set of required representations is denoted by $Rq(m)$ and the set of provided representations is denoted by $Pr(m)$.

Whenever a module specifies to provide a representation using the PROVIDES macro explained in section 3.1, it must implement an update method that fills the representation. That makes the update methods the smallest units that can be executed individually and are therefore considered as tasks from now on. Subsequently, the task graph $G := (V, E)$ with vertices $V := R$ and edges $E := E_{dep} \cup E_{upd}$ can be generated as follows:

$$E_{dep} := \{(r_a, r_b) \in V \times V \mid \exists m \in M : r_a \in Rq(m) \wedge r_b \in Pr(m)\} \tag{5.1}$$

The equation 5.1 for the graph edges ensures that all dependencies between modules and representations are met. For each pair (r_a, r_b) of representations, the update for representation r_a has to be executed before the update for representation r_b iff there is a module $m \in M$ that requires r_a and provides r_b. That models the same situation present in the framework and ensures the correct execution order.

Furthermore, the different update methods of a single framework module are often not thread safe because they may have side effects. Module developers can put data into class attributes that can be accessed from different update methods. That way, information that is generated by one update method can be used by the others of the same module. This is possible because the framework guaranteed the sequential execution of each update method due to the execution inside a single thread. The parallel execution may violate this assumption and lead to race conditions. To avoid that, a sequential execution of update methods for framework modules that are not thread safe, has to be ensured.

$$E_{upd} := \{(r_a, r_b) \in V \times V \mid \exists m \in M : r_a <_m r_b\} \tag{5.2}$$

Therefore, equation 5.2 adds dependencies between the provided representations of the same module m using a known strict total order relation $<_m$. However, a sensible order $<_m$ of update methods must be considered for best performance. Because the number of modules requiring a given representation varies, it seems reasonable to first provide representations that have a high number of dependent modules. That increases the count of ready tasks and thus increases the processor utilization to decrease the overall execution time. We investigate this assumption and the details on that further in section 6.3.2.

The conversion from the module-representation graph in figure 3.1 to task graphs is exemplary shown in figure 5.1. Each representation is converted to a task vertex and the dependencies between modules are resolved to dependencies between tasks that are shown as red edges. Furthermore, the dependencies between update methods of the same module are shown as blue edges.

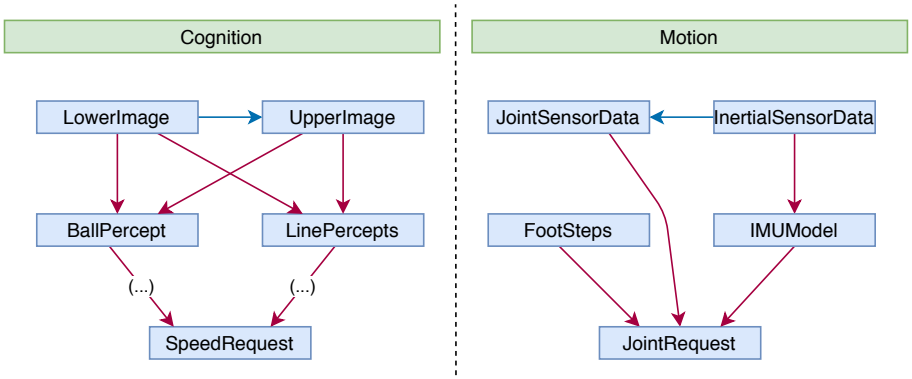


Figure 5.1: Task graphs generated from the exemplary module-representation graph in figure 3.1 showing E_{dep} edges in red and E_{udp} edges in blue.

5.2 Class design

As already mentioned at the end of section 4.3, the necessary prioritization of the motion cycle over the cognition cycle requires to run two instances of Taskflow each having a separate pool of threads inside a single cycle.

Therefore, the class design of the framework was changed and is shown in figure 5.2. It consists of the three main threads called Motion, Cognition, and Debug that are implemented in individual classes. While Motion and Cognition execute the processing cycles whenever new data arrives using the ModuleManager class, the Debug thread mainly handles the message transfer of debug data from the Motion and Cognition cycles via network to a connected computer. Before the changes, the Motion and Cognition classes inherited directly from the Thread class. The Thread class contains necessary objects that are needed for the execution like the Blackboard and objects for infrastructure and debugging explained in sections 5.3 and 5.4. The new design introduces an intermediate class SuperThread between Cognition/Motion and Thread that extends the features of the Thread class to contain multiple instances of the SubThread class. The SubThread class is instantiated by an initial Taskflow task and then passed to the SuperThread object.

We choose this design for the following reasons:

- The most important and central data structures of the framework, for example, the blackboard and other debugging objects contained in the Thread class, are also accessible via thread local global pointers. These pointers are initialized by the Thread constructor after the object creation. They allow an easy access to debugging and configuration features regardless of the position in code. To keep the functionality the same, each new thread has to instantiate the Thread class first, which is achieved by inheriting SubThread from Thread.

- While some data structures, for example, the blackboard, are shared between different threads of a cycle, others, for example, the message queues for debugging, are

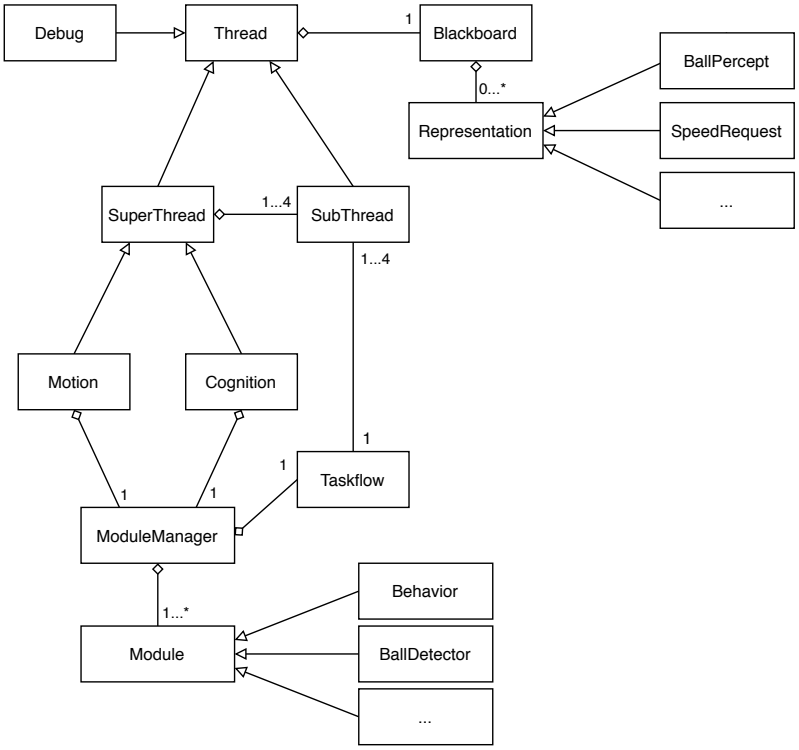


Figure 5.2: Simplified class diagram showing the relationship between representations, modules, and threads.

not. The SubThread and SuperThread classes take care of the required communication modifications in contrast to the traditional Thread class. In short, this design ensures that the SubThread's Blackboard instances are equal to the SuperThread Blackboard instance and the SuperThreads collect and distribute debug messages to and from the SubThread instances. The details on this issue are explained in section 5.3.

Using this design, the data processing loop works as follows:

1. The framework creates 3 threads and instantiates the classes Motion, Cognition, and Debug in each one.

2. The Motion and Cognition constructors instantiate the ModuleManager.

3. Each ModuleManager reads its configuration file containing the enabled modules and representations and calculates the task graph according to section 5.1.

4. The Cognition thread waits for new image data from the cameras and the Motion thread waits for a sensor update from the NAO body.

5. As soon as new data is available, the ModuleManager starts the cycle. If the thread pool is currently empty which is the case during framework start up, the

ModuleManager makes Taskflow to start new threads and instantiates the SubThread classes. Once that is done, the previously generated task graph is passed to Taskflow and executed.

6. During the execution of the cycle, the ModuleManager blocks the SuperThread and waits for completion.

7. The SuperThreads forward messages that have to be exchanged with other threads, which is explained hereafter in detail.

8. The Motion thread sends generated data to the actuators and the Cognition thread requests a new camera image.

9. If the module configuration was changed via debug command, the execution continues at step 3, otherwise, the execution continues at step 4 until the framework is stopped.

This design ensures that all necessary data structures for the module execution are still available and that the modules can access them in the same way as before.

5.3 Communication

In the past, the communication between different threads was realized via message queues. The message queues store and forward messages asynchronously and ensure data consistency without locks. They are used in many places of the framework and play a major role in the transfer of representations between Motion and Cognition and for debugging as already outlined in section 3.1. However, because of the needed copy operations, they should be used for small data only and cannot be used efficiently to communicate between sub threads that, for example, need to share image data as well.

To overcome this limitation, we changed the communication design. The new approach is displayed in figure 5.3. It shows 4 sub threads CT1, CT2, CT3, and CT4 for the super cognition thread and 2 sub threads MT1 and MT2 for the super motion thread as well as another thread for debug communication to the PC. The two blackboards containing all representations is used as shared memory for all threads within the motion and cognition cycles and is shown as gray ellipses. This allows multiple threads to operate on representations simultaneously while the task graph described in section 5.1 ensures data consistency. The task graph makes sure that the update method of a module that provides a representation executes before any other modules that require the same one. This leads to a memory access from multiple readers but only one writer that can be performed safely without data races.

Additionally, the different message queues are displayed as well. Representations that are shared between all 5 cognition and 3 motion threads are exchanged at the end of each

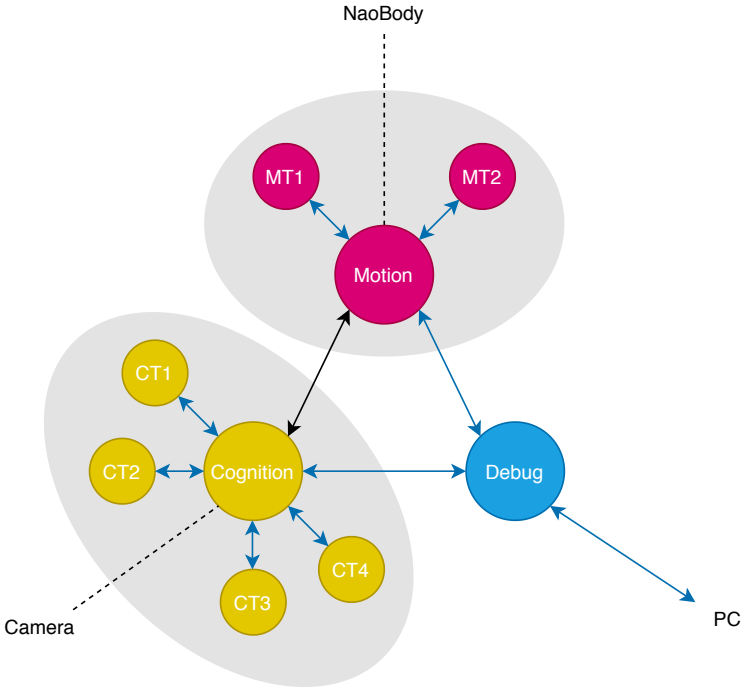


Figure 5.3: Communication between the Cognition, Motion, and Debug threads shown as yellow, red, and blue circles, respectively. Message queues are shown as arrows and shared memory regions are visualized as gray ellipses.

cycle by the two super threads via the message queue displayed as black arrows connecting them.

However, the representation transfer is not the only required communication. Debugging features also play an important role in the framework. Therefore, the user is able to send and receive debug messages using its PC that may get processed or generated by modules during runtime. The messages are used to display, visualize, or modify values of modules and representations and are very helpful to investigate the internal state of the robot for debugging. These messages are passed via message queues that are shown as blue arrows. The addition of sub threads also requires the addition of a new message queue for each sub thread that communicates with its super thread. This has the reason that multiple threads cannot access the same queue concurrently. Therefore, the sub threads first send their messages to the super threads which collect all received messages and forward them to the debug thread and vice versa. Since debug messages are only used during the development and not during actual games, the additional overhead is negligible and does not affect the performance during the game.

5.4 Debug classes

Besides the already mentioned debug messages, the framework also uses some debug classes that were used exclusively in each cycle to prepare and store data needed for remote debugging. This includes, for example:

- The TimingManager keeps track of the execution time of each module.
- The DrawingManager provides an interface for modules to draw debug information into, for example, camera images.
- The DebugRequestTable allows to execute code segments inside modules via command.
- The DebugDataTable stores arbitrary data that may be provided via commands by the user and can be used by modules.

The instances of these classes were stored in the Thread object and used during the execution of each module. When using multiple threads that access these objects concurrently, it could lead to data race conditions due to the internal use of non-thread safe data structures that may result in unspecified behavior. To overcome this limitation, three different solutions were individually found:

- *Read-only access* in sub threads for objects that are only modified by the super thread before or after the execution of each cycle, for example, the DebugDataTable. During the module execution, the objects are read-only and thus thread safe. This way, the same instance can be shared between the sub threads.
- *Separate instances* for objects that are modified during the execution of individual modules but are easy to duplicate, for example, the DebugRequestTable. The sub threads operate on their own instances and the super threads merge the objects at the end of each cycle.
- *Shared locks* for objects that are modified during the execution of individual modules but would have to be adapted for duplication beforehand, for example, the TimingManager and DrawingManager. They cannot be merged easily without changing the implementation details, which has not been covered yet. Shared locks are introduced as a temporary workaround until a better implementation is developed. Since the locks are mainly necessary for initialization purposes, the performance impact should be negligible and is covered by the evaluation in chapter 6.

5.5 Additional features

In addition to the main functionalities described before, we introduced a few more optimizations and additional features to the framework.

Subflows As already mentioned in section 4.3, Taskflow also supports the scheduling of dynamic tasks that are generated during runtime. Therefore, a new macro HAS_SUBFLOW, which can be used in the module definition presented in section 3.1, was introduced. Similar to the PROVIDES(...) macro, it forces the programmer to implement an update method. Instead of a reference to a representation that has to be filled, a Subflow object is passed. The Subflow object makes the Taskflow API directly available and allows the generation of sub task graphs that are treated like single tasks. This method is called before any other regular update method of the module and it allows to parallelize program code inside modules easily without, for example, spawning additional threads manually.

```
1  void YoloRobotDetector::update(tf::Subflow& subflow)
2  {
3    subflow.emplace([=]()
4    {
5      execute(true);
6    }).name("YoloUpper [YoloRobotDetector]");
7
8    subflow.emplace([=]()
9    {
10     execute(false);
11   }).name("YoloLower [YoloRobotDetector]");
12 }
```

Listing 5.1: YoloRobotDetector subflow generation.

This mechanism is implemented exemplary for the YoloRobotDetector in listing 5.1, which executes a neural network for image segmentation. Using this feature, both images can be processed concurrently in the same module. Lines 5 and 10 execute the inference of the neural network for the upper and lower camera using lambda functions that are added as tasks in lines 3 and 8. Once both functions have been executed, the regular update methods of the module provide the representations using the results of the inference done before.

Concurrent updates The task graph generation described in section 5.1, assumed that update methods of the same module cannot be executed concurrently. This is true for many modules but not for all. For the case that an update method can be executed independently from the others, we introduced the new macro PROVIDES_CON-

CURRENT(...). The functionality is similar to the regular PROVIDES(...) macro, however, this update method is not considered when generating the strict total order of representation updates $<_m$ for this module. That allows the concurrent execution of update methods if they do not operate on shared data or if they access it thread safely.

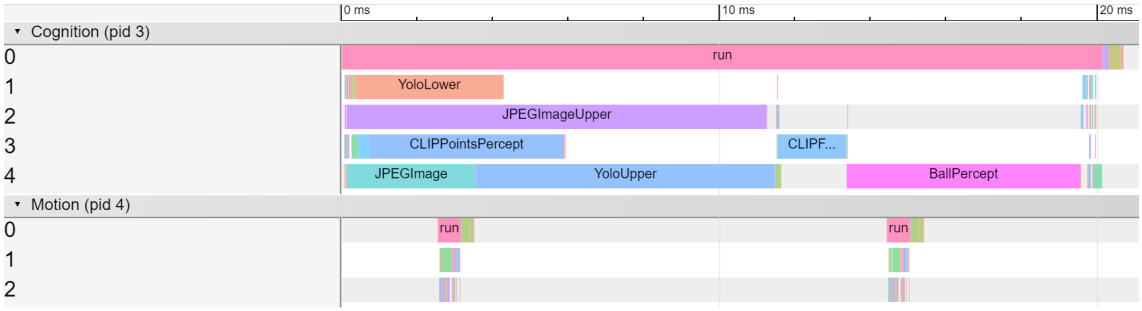


Figure 5.4: Execution trace of 3 motion threads (1 super thread and 2 sub threads) and 5 cognition threads (1 super thread and 4 sub threads).

Task trace generation One of the main advantages of Taskflow outlined in section 4.3, is the ability to observe and record the execution order of tasks in real-time, which can be interactively viewed via Google Chrome's tracing feature[1] and is shown in figure 5.4. The image shows the exact scheduling of individual tasks to a number of threads that was recorded during the execution of a single cycle. However, we overhauled the implementation by Taskflow completely during the framework integration in order to implement the following features:

- Parallel observation of multiple task graphs. Because the motion and cognition execution runs concurrently, it is helpful to see the timeline of both cycles.
- Observation of the super thread that is run outside of the task graph.
- Additional time measurements to monitor, for example, the message queue communication that happens before and after the execution of each cycle.

This feature is very helpful during the analysis of possible performance bottlenecks and it allows deep insights into the framework that were not visible before. The recording can be started and stopped at any time in real-time on the robot using two debug commands and the measurements get transferred into a file of the connected computer automatically. This file can be opened directly using the Google Chrome tracing tool. That allows to have a quick glance into the module scheduling for developers, if, for example, the cycle execution time suddenly takes longer than they expect.

[1] `https://www.chromium.org/developers/how-tos/trace-event-profiling-tool`

5.6 Summary

This chapter first explained the transformation from a module-representation graph to a task graph. Therefore, we considered the individual update methods as tasks and transformed the dependencies based on the module requirements to task dependencies. After that, we presented the required changes to the class design of the framework and highlighted the occurred challenges concerning the thread communication. Furthermore, we explained some required changes to the debug classes that were not thread safe yet and presented the addition of useful features to improve the degree of concurrency and ability to debug.

We evaluate the impact of these changes on the overall performance of the framework in the following chapter.

Chapter 6

Evaluation

In this chapter, we evaluate the impact of the changes explained earlier and measure the speed up gained by the parallelization. Therefore, we examine the execution time of the motion and cognition cycle under different scenarios and analyze the strengths and weaknesses of the found solution.

6.1 Metrics

The main goal when evaluating the performance of the framework is to meet the deadlines of the motion and cognition cycles. This means that for cognition, the execution time after receiving new image data must be below 33 ms when targeting a frame rate of 30 Hz and for motion, the execution time after receiving new sensor data must be below 12 ms when targeting a frame rate of 83 Hz.

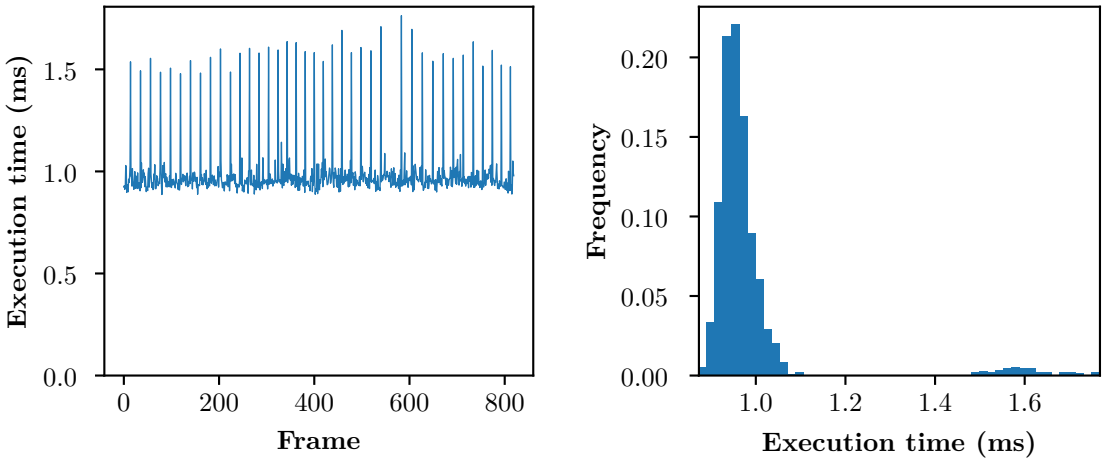
However, for both cycles, new image and sensor data are triple buffered. Whenever a cycle needs more execution time than usual, the framework caches new data that arrives during the ongoing execution and keeps it for the following cycle. That avoids discarding data and reduces the consequences of missed deadlines. Nevertheless, as already outlined in section 2.2, the impact for motion is higher than for cognition. If the motion cycle misses a deadline for a new actuator request, the robot may stutter while a small delay in cognition processing may not even be noticeable. That gives the monitoring of deadlines a higher priority for motion than for cognition. However, focusing only on missed deadlines is insufficient when comparing the performance of different configurations or implementations that both finish within their deadlines. Furthermore, when monitoring the performance of the processing cycles, we need to consider that these may also be affected slightly by lower priority logging and debugging threads of the framework as well as other software by the manufacturer running on the NAO. The latter includes kernel drivers and the high priority daemons necessary for hardware communication and message transfer, which are part of the NAOqi control software, and further Linux typical logging, audio, and network

services. When the framework is stopped and the robot idles, these processes require about 3 % to 5 % CPU usage on each core.

Consequently, aside counting the number of missed frames and printing a warning whenever the buffer is full and input data is discarded, we monitor the timestamps of each cycle start and finish and consider the difference as the execution time. For the time measurements, we use the std::chrono::steady_clock class from the C++ Standard Library. During the experiments outlined in the following sections, we record all execution times of each cognition and motion cycle over a longer period of time and calculate the following metrics on them:

- The *average execution time* is the arithmetic mean of the measured execution times during the experiment. We use it to evaluate the default case under the condition that the variations are small enough.
- The *0.01 and 0.99 percentiles* indicate the distribution of the measured execution times. The execution time must be as stable as possible to utilize all resources of the processor while reducing the chance to exceed the deadlines too frequently. The difference between both percentiles contains 98 % of all values and indicates the stability of the execution time. Bad scheduling that happens too frequently causes a wider distribution, which increases this difference.
- The *maximum execution time* is the longest execution time during the experiment. It is used to estimate the worst case execution time, which must not exceed the deadline for motion and is accepted to exceed the deadline for cognition when it happens rarely. However, because of other processes and interrupt handlers that are running on the same operating system, some occasional high execution times happen from time to time that are out of control of the Nao Devils framework. Hence, the 0.99 percentile is more meaningful when evaluating the worst case execution time that is under control of the framework.

During the evaluation of the motion cycle's execution times, we discovered that these vary systematically and independently from external influences. This is because the execution time for motion depends on the current step phase. In the Nao Devils framework, one foot step consists of several frames that do the required motions for lifting, moving, and setting down each foot. Between two steps, the walking engine recalculates its preview, which contains the following three planned steps that are based on the sensor data gathered before. This recalculation happens periodically in a single frame and is quite CPU-intensive, which increases the execution time significantly. Figure 6.1(a) shows the execution times of a 10 s recording and visualizes the spikes. However, the occurrence of these spikes may pause when the robot stops walking shortly to prevent a fall down or during the performance of kicks, which is visible around frame 550. On the one hand, these



(a) Execution time of each frame within a 10 s time span.

(b) Histogram of execution times using 50 bins.

Figure 6.1: Fluctuations in motion execution time.

spikes increase the difference between the 0.01 and 0.99 percentile a lot while not being caused by scheduling decisions or parallelization, which we want to measure primarily using this metric. On the other hand, the number of preview frames depends on the number of kicks. The more often the robot kicks the ball or prevents a fall down, the lesser the robot recalculates its preview. That makes all metrics dependent on the actual number of preview frames. To eliminate this dependency, we separate both cases and evaluate them independently. The average execution time of 1.628 ms of preview frames is around 69 % higher than non-preview frames executing 0.9646 ms on average. Figure 6.1(b) shows a histogram that plots the frequency of different execution times grouped into 50 bins. It shows that there are no frames having an execution time around 1.3 ms. Consequently, we separate the motion times right in the middle between the two local maxima into preview and non-preview frames. That allows a more precise evaluation of metric changes and an independent evaluation of both cases.

6.2 Setup

The setup used for the evaluation must fulfill three requirements:

1. The framework's performance must be analyzed on the robot itself and cannot be simulated. The execution time of a program depends on the used CPU and operating system. Since the NAO runs an Intel Atom CPU, which we introduced in section 1.2 and which is uncommon for desktop computers, the same measurements cannot be produced on a different system. Furthermore, when using the simulator

explained in section 3.1, the framework replaces hardware dependent modules for the communication with the cameras, actuators, and sensors by others that have different performance characteristics.

2. The setup must reflect a typical game situation. For example, the execution time of the ball detection depends on the number of ball hypotheses, which are generated heuristically during the image preprocessing. For each hypothesis, a small neural network decides whether it is a ball or not. Since the number of hypotheses varies, the execution time for the ball detection varies as well. Especially robots generate many false ball hypothesis because of their similar black-and-white look and have a high impact on the execution time of the whole cognition cycle. A realistic game scenario helps to keep the execution time in a realistic range.

3. The setup must be reproducible. Since we compare the performance of the framework before and after the added parallelization as well as we test different configurations, every run of the same configuration must produce the same results. This ensures that differences in measurements, for example, of the average execution time, are due to the changed configuration and not caused by other external influences.

Consequently, two setups are possibly suitable:

First, the robot replays a log file recorded previously. Since the execution time depends on image and sensor data, it is theoretically possible to replay some data that we captured earlier. This way, the processing cycle receives the recorded data and ignores the real data from the robot. That makes the robot operate without perception of its real environment. This has the advantage of being very easy to reproduce and does not require a field setup with other robots and a ball. However, while sounding promising, this setup has a few issues.

Especially the loading and decompression of image data needs additional execution time and memory. On the one hand, if we decide to load the image data before the execution of each cycle, it will affect the measured execution time. On the other hand, if we decide to load all image data once during the framework start, the number of images will be limited by the available memory. Furthermore, we are unable to record log files during games that contain every image in its full size. Either, we record at a greatly reduced frame rate or resolution for debugging purposes or we record short image sequences of two seconds periodically for the training of new neural networks. The recording of complete games during the normal execution of the framework exceeds the available computing power or storage and is currently not possible.

Second, we set up a typical game situation, start the robot from the same position at the field border each time, and let it move the ball to the opponent's goal as shown in figure 6.2, which takes around 30 s. This has the advantage that there is no need to

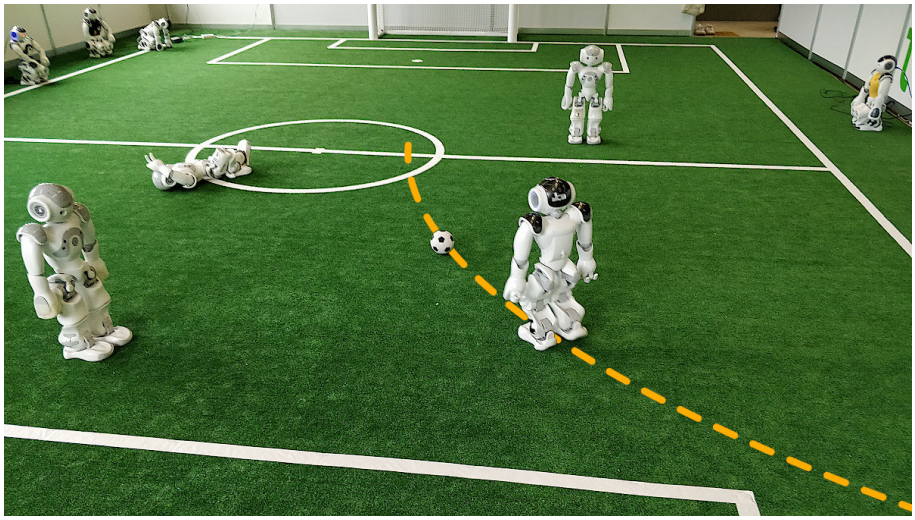


Figure 6.2: Evaluation setup with a NAO walking along the orange dashed line.

modify anything on the robot for the evaluation and it can operate as normal as during a game. In the following sections, we analyze the reproducibility of the explained live evaluation setup, we show the difference to recorded log data, and we point out that the live evaluation results in more realistic execution times.

For the following preliminary evaluations, we use the original framework without any multi-core changes.

6.2.1 Reference measurements

Metric	Minimum	Average	Maximum
p=0.01	12.34 ms	12.93 ms	14.09 ms
avg	15.88 ms	16.09 ms	16.18 ms
p=0.99	18.85 ms	19.64 ms	21.02 ms
max	20.03 ms	22.25 ms	24.97 ms

Table 6.1: Reference measurements of cognition metrics during 10 test passes.

	Non-preview			Preview		
Metric	Minimum	Average	Maximum	Minimum	Average	Maximum
p=0.01	0.8950 ms	0.8997 ms	0.9040 ms	1.494 ms	1.503 ms	1.520 ms
avg	0.9617 ms	0.9646 ms	0.9668 ms	1.623 ms	1.628 ms	1.633 ms
p=0.99	1.064 ms	1.080 ms	1.101 ms	1.784 ms	1.835 ms	1.924 ms
max	1.113 ms	1.182 ms	1.265 ms	1.802 ms	1.887 ms	1.997 ms

Table 6.2: Reference measurements of motion metrics during 10 test passes.

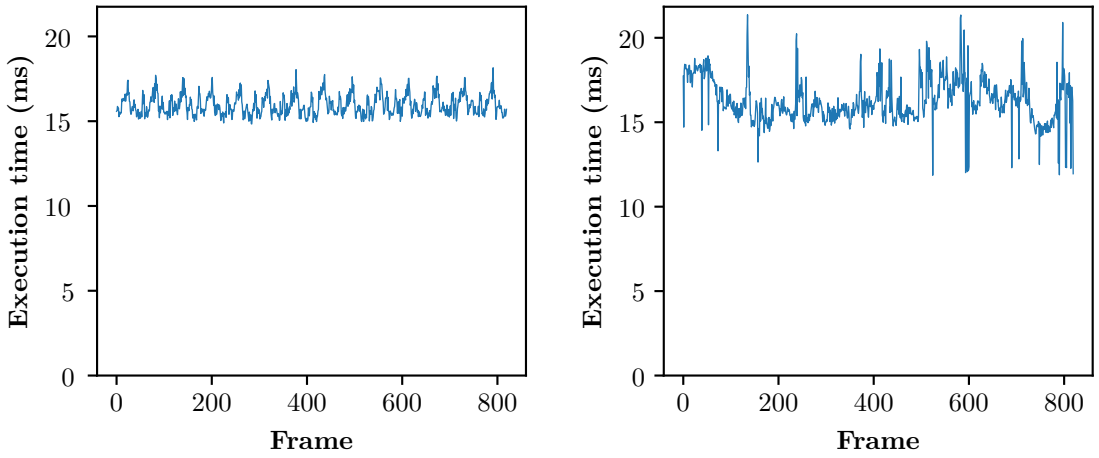
In order to be able to make precise statements about changes in metrics that are caused by the introduced multi-core implementation, we first analyze the performance of the old framework and investigate the reproducibility of the presented evaluation setup. Therefore, we use the original Nao Devils framework without any multi-core improvements, except for a bugfix concerning the cognition real-time priority that we discuss in section 6.2.3. We use the same framework configuration as for official games. Moreover, in the following evaluations, we use the results as reference measurements whenever we compare the outcome of an experiment to the previous, non-parallelized framework. We repeat the same setup 10 times, calculate the metrics for each pass, and show the average, minimum, and maximum values of these during all 10 test passes in tables 6.1 and 6.2 for cognition and motion, respectively. Since the minimum and maximum values deviate from the average value by a maximum of 9 % for the percentiles and by a maximum of 1.3 % for the average metric, we will refrain from testing the same configuration several times in the following experiments and consider the results as reproducible. Consequently, we interpret measurements outside of these recorded ranges as caused by framework improvements and configuration changes that we investigate in detail hereafter.

6.2.2 Comparison between log and live data

	Log data	
p=0.01	15.04 ms	(+16.32 %)
avg	16.05 ms	(−0.2681 %)
p=0.99	17.77 ms	(−9.550 %)
max	18.15 ms	(−18.42 %)
98% range	2.722 ms	(−59.43 %)

Table 6.3: Cognition metrics using log data and their deviations from the reference measurements using live data.

We compare the execution times using a two seconds log file sequence from RoboCup 2019 in Sydney with the live scenario on the field. The log file plays in a continuous loop during the evaluation while the framework's actuator output is ignored and the robot's motors are turned off. The motion execution times are unaffected by this change, which is reasonable because they are independent from any sensor input and mostly use algorithms with constant runtime. However, the number of preview frames varies because of missing stumbles, but these are already separated using the data split explained in section 6.1 and do not change the results. For cognition, we show the calculated metrics based on the recorded execution times in table 6.3. The average execution time is very close to the reference measurements using live data. Nevertheless, the fluctuations in execution time for cognition using log data are much less than on the field. Using log data, 98 % of the



(a) Log data.

(b) Live data.

Figure 6.3: Comparison between cognition execution times using recorded log data and live data on the field.

cognition execution times are distributed over a 2.722 ms range. In contrast to that, the execution times using live data are distributed over a 6.709 ms range. These fluctuations are also visible when observing the raw execution times of each frame, which we show in figure 6.3. The execution times using the 2 seconds log file follow a very repetitive pattern. In contrast, the execution times during the live test vary much more and cover extreme values that are not reached using log data. These differences are of particular importance when considering the performance of the implemented changes and investigating the edge cases.

Consequently, we decided to use the more realistic field evaluation for the following investigations.

6.2.3 Real-time priority

	Non-real-time	
p=0.01	13.34 ms	(+3.157 %)
avg	17.46 ms	(+8.506 %)
p=0.99	22.80 ms	(+16.10 %)
max	26.95 ms	(+21.13 %)
98% range	9.463 ms	(+41.06 %)

Table 6.4: Cognition metrics before setting fixed real-time priority and their deviations from the reference measurements.

During the evaluation, we discovered a misconfiguration of the real-time priorities used in the Nao Devils framework. Usually, the priorities of the different processes running on the operating system are ordered according to the data flow of actuator and sensor data. Processes provided by the manufacturer that communicate directly with the hardware have the highest priority in order to receive and send motor and sensor updates in time. After that, a small program with slightly lower priority follows that connects to the API provided by the manufacturer and communicates with the Nao Devils framework. The framework's motion thread again uses an even lower priority while setting the cognition thread to the lowest real-time priority. The debug thread and all other processes are scheduled by the standard time-sharing scheduler. However, for the cognition thread, the real-time priority was only set while waiting on new image data from the cameras. As soon as these were available, the priority was set back to Linux' standard time-sharing scheduler. We removed these priority changes and set it to a fixed real-time priority instead. For motion, the impact of this change is marginal, which is expected because motion is always prioritized over cognition. In contrast, for cognition, this makes a major difference. We show the impact of these changes in table 6.4. The average cognition execution time is 8.5 % higher and the 0.99 percentile is even 16.1 % higher without a fixed real-time priority. Consequently, keeping the real-time priority permanently for the cognition thread reduces the average execution time significantly and decreases the 98 % value range from 9.163 ms down to 6.708 ms.

This behavior is comprehensible when examining the Completely Fair Scheduler (CFS), which Linux uses for non-real-time tasks by default. The CFS tries to assign each running process the same amount of computing time. Therefore, CFS tracks the elapsed runtime of each process and always schedules the process with the minimal execution time so far during each timer interrupt [16]. However, this strategy disadvantages the cognition part of the framework and reduces the responsiveness. Since cognition utilizes most of the resources of the whole system, it is the first process that is preempted whenever one of the other processes running at the operating system is ready to execute. That introduces overhead for the process switches and increases the cognition execution time non-deterministically.

Hence, since all other processes running on the NAO are less important and time-critical than the cognition processing, this is considered a bug in the current framework. Consequently, we prioritized cognition over the standard Linux processes using the lowest real-time priority for all evaluations.

6.3 Scenarios

In this section, we evaluate different configurations for the number of threads for cognition and motion, for the order of update methods mentioned in section 5.1, and for different

thread to CPU core assignments. Furthermore, we stress test the implementation by enabling additional framework modules that utilize more CPU resources.

In the following, whenever we refer to the number of motion or cognition threads, we actually refer to the number of working sub threads from section 5.3 that execute the framework modules and assume an implicit additional super thread being present in any case that blocks during the module execution.

6.3.1 Thread count

In section 4.3, we decided to use two separate thread pools to prioritize the motion over the cognition threads. However, using this approach, we must decide how many threads are spawned for each cycle in order to use all CPU resources while avoiding unnecessary context switches when more threads are ready to run than CPU cores are available. Therefore, we do 16 tests with 1 to 4 cognition and 1 to 4 motions threads and investigate the impact on the execution times.

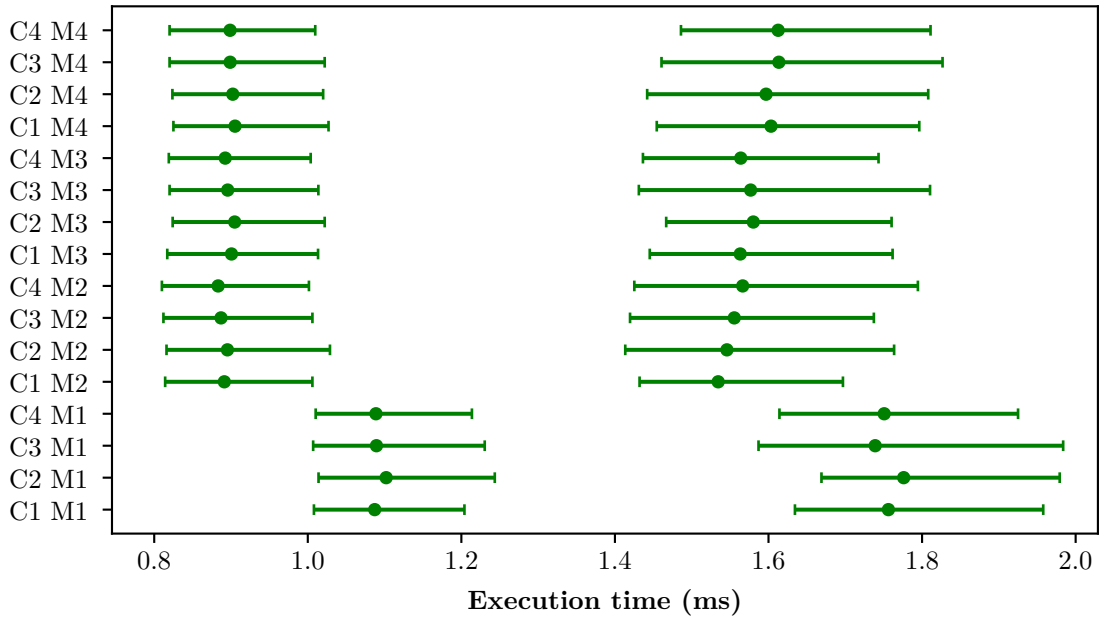


Figure 6.4: Average execution times and the 0.01 and 0.99 percentiles of the *motion cycle* when using different numbers of threads for motion and cognition.

First, we analyze the average execution time of the motion thread and show the average execution times and the percentiles in figure 6.4. The measurements around 1 ms are non-preview frames and the measurements around 1.6 ms are preview frames. The number of cognition threads does not have any measurable, significant impact on the execution time of the motion cycle because of its higher priority. When considering the number of motion

threads, the use of only one thread is clearly the slowest option consuming more than 1 ms of execution time for non-preview frames and more than 1.7 ms of execution time for preview frames on average. The execution times for 2 to 4 threads are very similar and only increase slightly for the preview frames using 3 or 4 threads. We expected this result because most motion modules are very short and do not allow massive parallelization because of their dependencies.

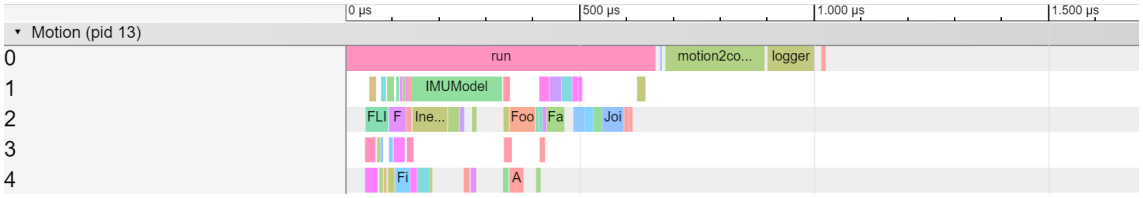


Figure 6.5: Execution trace of 4 motion threads for a *non-preview frame*.

Figure 6.6: Execution trace of 4 motion threads for a *preview frame*.

Figure 6.5 shows the execution trace using 4 threads and illustrates the problem. Especially thread 3 and 4 do not have enough work to do and the execution time is mostly limited by modules that require a sequential execution. During a preview frame, which we show in figure 6.6, this effect becomes even more visible because the preview calculation is performed within a single module providing the FootSteps and all other threads must wait for its termination. Furthermore, Taskflow follows a hybrid approach between busy-waiting and blocking whenever a thread does not have any tasks to execute. First, the idle thread tries to steal tasks from other task queues up to 100 times. If this does not succeed, it will block the execution. That increases the execution times for non-preview frames even further using 3 or more threads. Whenever a thread blocks, the rescheduling requires a small amount of time whenever there is new work to do and extends the total execution time. For example, figure 6.6 shows that the module execution in threads 1 and 2 is slightly delayed after providing FootSteps, although they only depend on the FootSteps representation.

Subsequent, we decide to use 2 threads for motion, which offers the best performance during the experiment. Furthermore, we do not expect that the motion processing requires much more computing power in the near future and we can use the available resources for cognition more effectively.

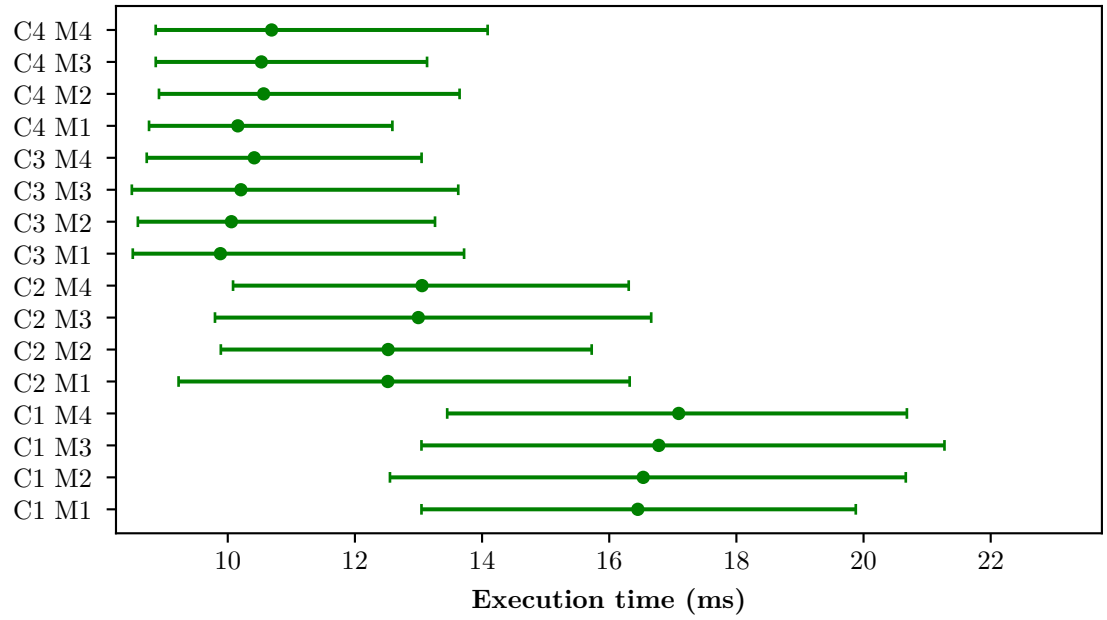


Figure 6.7: Average execution times and the 0.01 and 0.99 percentiles of the *cognition cycle* when using different numbers of threads for motion and cognition.

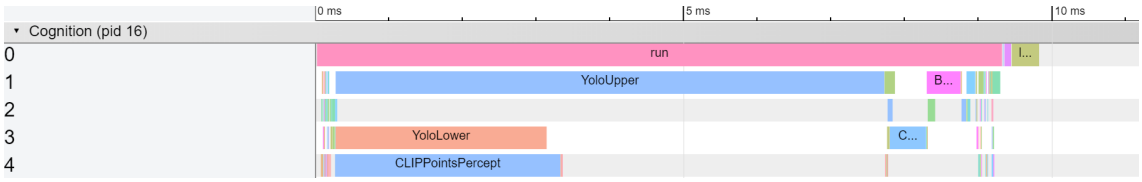


Figure 6.8: Execution trace of 4 cognition threads.

For the cognition execution times, we show the averages and percentiles in figure 6.7. Here, the impact of different motion thread counts is more visible. Since the motion cycle preempts the cognition cycle whenever new data is available, the usage of more motion threads adds more overhead for context switching and increases the cognition execution time slightly. From the cognition perspective, one motion thread is ideal, but since this choice increases the execution time for motion more than it decreases the execution time for cognition, using two motion threads is the best compromise.

When considering the number of cognition threads, 3 or 4 threads are the best choice offering minimal execution time. However, when using 4 threads, the same effect as mentioned in the motion evaluation becomes important. Figure 6.8 shows the execution trace using 4 cognition threads and illustrates only 3 major modules running concurrently, while many threads idle most of the time. Here, Taskflow blocks idle threads after some time as well, which introduces a small overhead whenever they must be rescheduled. Consequently, 3 threads are able to process the workload in the same time while reducing these

overheads whenever no modules are ready to execute. That reduces the total execution time compared to 4 threads. Nevertheless, we decide to use 4 cognition threads for the following reasons:

1. The execution time difference between 3 and 4 cognition threads is small compared to the total runtime. When using it in combination with 2 motion threads, the configuration of 3 cognition threads reduces the average execution time from 10.56 ms to 10.06 ms. Since both times are much smaller than the deadline of 33.33 ms to reach the frame rate of 30 Hz, this is not of particular importance.

2. The Nao Devils team plans to add more complex modules in the near future that will use the available resources, for example, for improved image and audio processing. We evaluate the impact of some more demanding modules on the execution time in section 6.3.3 to investigate the performance under heavier load.

3. Other developers working on the framework do not have the knowledge when to switch between 3 or 4 cognition threads. Since a requirement for the implemented solution is the avoidance of configuration parameters, using 4 threads right from the beginning makes the framework ready for the future without manual adjustments.

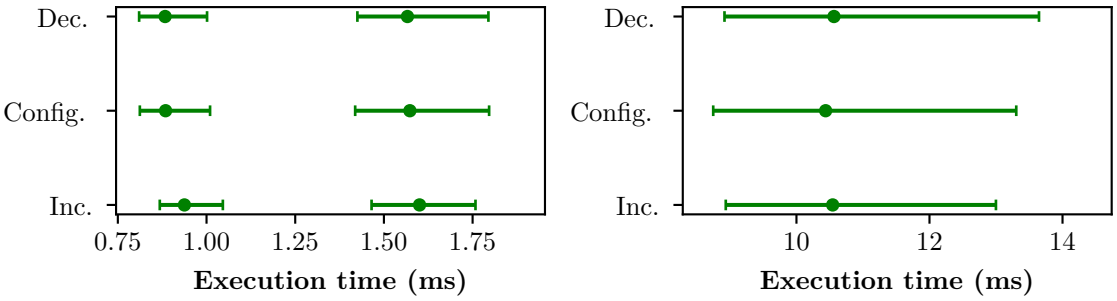
Consequently, we decide to use 2 motion and 4 cognition threads as the best choice for the following evaluations.

6.3.2 Update order

In section 5.1, we mentioned the total order relation $<_m$ that defines the sequential order of update methods providing different representations within the same module to avoid race conditions on shared data. We argued to execute update methods first that provide representations having a high number of dependent modules to increase the number of ready modules as early as possible and thereby improve parallelism. Therefore, for each update method of a single module, we first calculate the number of all subsequent nodes and then add additional dependencies into the task graph to execute these methods in *descending order*.

This evaluation compares this choice with two alternative orderings. First, we use the unmodified order of the module manager's configuration file that is sorted alphabetically according to the names of the representations, which is the default case of the internal data structure when no explicit sorting is performed. Second, we reverse the explained order by sorting the update methods in *ascending order* of dependents. We consider this as the worst case because it reduces the number of ready tasks at the beginning of the cycle and delays the efficient parallel execution to the end of it.

We evaluate the three settings and show the results in figure 6.9. For cognition, the differences in the average execution times are very small compared to the total execution



(a) Motion cycle.

(b) Cognition cycle.

Figure 6.9: Average execution times and the 0.01 and 0.99 percentiles when ordering update methods by their increasing or decreasing number of dependents or by configuration file.

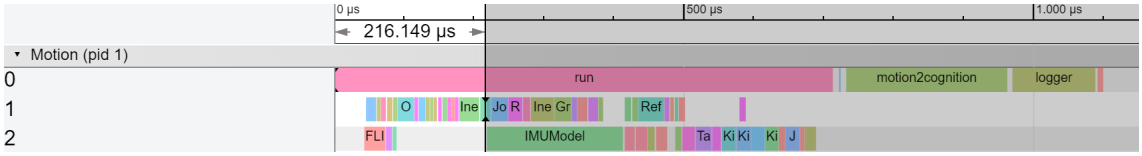


Figure 6.10: Execution trace of a non-preview motion frame with update methods ordered by their *increasing* number of dependents.

time. We measure the biggest difference of 0.1255 ms between the decreasing dependents ordering and the configuration file ordering. Furthermore, this difference is much smaller than the value range of the average execution time during the reproducibility tests in section 6.2.1. Consequently, this is not considered significant. For motion, the results are different. The differences between the decreasing and increasing dependents ordering are 54.4 µs for non-preview frames and 34.1 µs for preview frames in favor for the decreasing ordering, which is much more significant when considering the value range of the average execution time of the motion cycle during the reproducibility tests. This observation is reasonable, when considering the execution trace of a single motion cycle in comparison. Using the increasing dependents ordering in figure 6.10, the framework provides the FrameInfo representation, which is a major representation of the framework and required by many modules, only after more than 200 µs. That leaves the second thread idle between 100 µs and 200 µs, which reduces the parallelism and increases the total execution time.

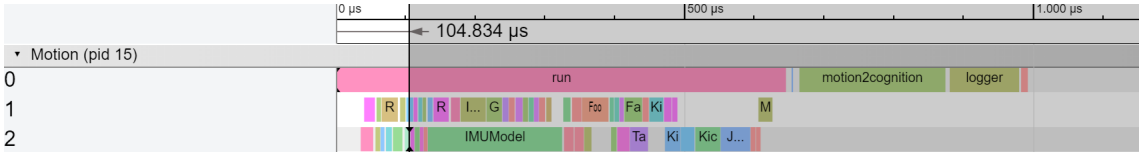


Figure 6.11: Execution trace of a non-preview motion frame with update methods ordered by their *decreasing* number of dependents.

In contrast, when using the decreasing dependents ordering showed in figure 6.11, the framework provides the FrameInfo representation much earlier and utilizes both threads more efficiently.

The measurements show that at least for motion, the total execution time depends on the chosen update order. However, for cognition, it is theoretically possible as well. The effect is less visible in this case because the total execution time is more dependent on the execution of a few CPU-intensive modules that start early enough in both configurations to be unaffected by the change.

Nevertheless, it is worth noting that the dependent-based ordering is a heuristic approach to improve the resource utilization, which is easy to implement but not optimal. It is based on the assumption that all modules roughly take the same execution time, which is obviously not the case in reality. A better solution is to prefer update methods that need much time to execute and to reduce the idle time of all threads. However, we do not have knowledge about the execution time of each module in advance and the optimal order may be hard to find. Further research on this problem may be the subject of an analysis at a later time.

In our case, the decreasing dependent ordering performed quite well and we did not encounter any high execution times being caused by an unsuitable execution order of update methods during all of our tests.

6.3.3 Stress test

The Nao Devils team ported the framework to the NAO version 6 just recently and all modules have been designed not to exceed the maximum execution time on a single core in the past. Consequently, it is currently not capable to fully utilize all available CPU resources until the team develops more complex modules in the future. To simulate more load on the entire system, we enable the JPEG image conversion for the upper and lower camera, which allows to transfer the image data to a connected computer in real-time over WLAN, and we enable the inference of an experimental neural network providing additional ball hypotheses. In sum, this adds about 20 ms of additional execution time.

Since this change only affects the cognition execution time, we ignore the motion data for this experiment and compare the performance of the old framework with the multi-thread implementation using different cognition thread counts from 1 to 4. We show the execution times in figure 6.12. The average execution times for the reference test and one cognition thread are over 33 ms and cause a high number of missed deadlines. The cognition cycle missed the deadlines in 7.493 % and 10.24 % of all frames, respectively. That makes this configuration unusable during a game. The usage of one cognition thread turns out to perform even worse compared to the reference test because of the additional thread communication and dynamic task execution overhead. However, when increasing

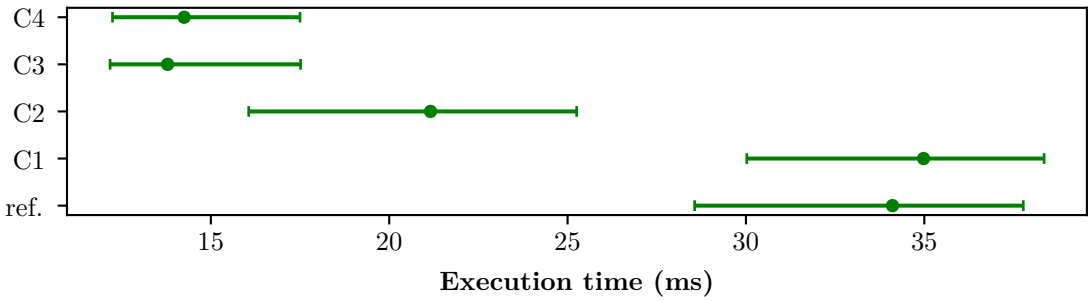


Figure 6.12: Average execution times and the 0.01 and 0.99 percentiles of the *cognition cycle* when using different numbers of cognition threads compared to the reference measurements of the non-multi-threaded framework.

the thread count to two or more, the average execution times decrease rapidly offering an improvement of 60.58 % when comparing the average execution time of one cognition thread to three cognition threads.

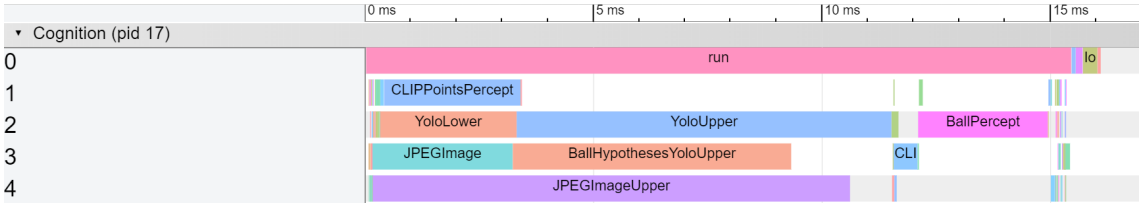


Figure 6.13: Execution trace of a cognition frame with enabled JPEG conversion and generation of additional ball hypotheses using 4 cognition threads.

Nevertheless, the four thread configuration still performs slightly worse compared to three. Figure 6.13 shows the execution trace using 4 cognition threads. Even with the additional CPU-intensive modules, four threads do not have enough concurrent work to do to fully utilize all cores. In contrast to section 6.3.1, the reason for that is not only the introduced overhead because of blocking tasks but it indicates another problem, which is the scheduling of tasks. The total execution time of the shown execution trace can be decreased by swapping YoloUpper with CLIPPointsPercept and executing the first one in thread 2 and the second one in thread 1. This closes the gap in threads 1, 3, and 4 at the time around 11 ms and reduces the overall execution time by about 1 ms. The cause of this execution order is the point in time when the dependencies for the execution of YoloUpper and CLIPPointsPercepts are fulfilled. Since YoloUpper and YoloLower are ready to start slightly earlier than CLIPPointsPercept, the execution of YoloLower starts first. Since thread 1 is busy at this time, the execution of YoloUpper postpones. Once thread 1 is free, the execution of CLIPPointsPercepts starts because it is a direct successor of a previous representation in the same thread that is executed directly by the work-stealing

scheduler to improve data locality. Consequently, the execution of YoloUpper is delayed until YoloLower finally finishes.

This scenario illustrates that the work-stealing scheduler used by Taskflow does not guarantee a minimal execution time. However, for our use case, we did not observe any major performance problems during the evaluations. Nevertheless, it becomes more important when the load rises and the number of long executing modules increases. To reduce the chance of bad scheduling decisions, we encourage developers to write a higher number of short modules instead of a single one that has a high execution time, which also follows the general design concept of the framework architecture. Alternatively, they can use the Subflow feature explained in section 5.5 to split the execution into smaller pieces whenever possible. Nevertheless, if developers cannot avoid to add new long executing tasks, they must verify the efficient scheduling and adjust the dependencies where necessary, for example, using the provided debug tool for execution tracing. In the future, further investigations may be needed to circumvent such situations.

However, a short test using another dummy module iterating for several milliseconds within an empty for-loop performed significantly better using four threads than three. Consequently, the final choice of four cognition threads is reasonable for the future when enough parallel workload will be available.

6.3.4 CPU core pinning

During all evaluations so far, we did not assign each thread to a fixed physical CPU core and trusted the Linux scheduler to take care of the distribution. However, it is possible that the execution time reduces by pinning each thread to a specific core in order to improve the CPU cache usage and reduce the overhead of possible thread migrations.

We evaluate this by comparing different thread to CPU core assignments. For each cycle, motion and cognition, we either set each thread to a fixed physical core or use the default dynamic assignment of the Linux process scheduler. Since we only use two threads for motion, we make sure to use two cores that share a common L2 cache. Since the impact on the execution time is very small, we set the robot to a fixed position on the field and disable all movements. That reduces the runtime fluctuations because of varying image data to a minimum. Furthermore, we also enable the JPEG image conversion and the experimental neural network for additional ball hypotheses like in the previous experiment to increase the load on the system, which makes the effect more visible. Since the 0.01 and 0.99 percentiles are unaffected by the changes and remain at the same positions relative to the average execution times, we do not cover them at this point. Furthermore, we ignore the motion preview frames for simplicity reasons because their execution times scaled proportional to the non-preview frames during the experiment.

Config.	Motion	Cognition
C. dyn., M. dyn.	0.9045 ms	13.93 ms
C. dyn., M. fix.	0.8885 ms	14.17 ms
C. fix., M. dyn.	0.8989 ms	13.82 ms
C. fix., M. fix.	0.8868 ms	13.88 ms

Table 6.5: Average execution times of the non-preview motion and cognition cycle using either fixed or dynamic thread to CPU core assignments.

Table 6.5 shows the average execution times using either a fixed or dynamic assignment of CPU cores for each cycle. We repeated the experiment several times and the values varied only slightly while the order kept the same each time.

We can only speculate about the reasons for the outcome of the experiments because the effects depend on CPU caching and scheduling decisions of the operating system that are hard to monitor.

When considering the motion cycle independently, the best choice is a fixed assignment of both cycles resulting in a minimal motion execution time. That is reasonable because the motion cycle benefits from a shared L2 cache that reduces access times to data. However, for cognition, a dynamic assignment of the motion threads is slightly faster. That is reasonable as well because whenever cognition currently uses at least one of the two CPU physical cores that the motion cycle is assigned to statically while other cores may be free, the prioritized motion cycle must preempt the cognition thread without necessity. That increases the total cognition runtime. The third option, a dynamic assignment of cognition and fixed assignment of motion threads raises the execution time for cognition even over the Linux default setting using dynamic assignments for both cycles. Using this configuration, the process scheduler must migrate running cognition threads to another core whenever the prioritized motion threads start the execution on the pinned cores, which introduces additional overhead.

When considering both cycles together, a fixed assignment of cognition and a dynamic assignment of motion threads delivers the best results during this experiment and reduces the execution times of the motion and cognition cycles compared to the default case by 0.6200 % and 0.7420 %, respectively.

However, since the benefit is very small and was not measurable during the regular experiments using a walking robot, the differences are negligible. Furthermore, the evaluation only covered the execution times of the Nao Devils framework. Neither did we measure the effects on other real-time processes, for example, for the communication with the robot hardware that may be affected by the changes as well, nor did we measure consequences for non-real-time tasks for debugging, logging, and other services.

Consequently, we decide not to use core pinning for the framework in order to keep the flexibility of the Linux process scheduler and to avoid negative impacts on other processes that we are not aware of.

6.3.5 Reference comparison

In this section, we compare the final implementation using the results from the previous evaluation with the reference measurements from section 6.2.1. Therefore, we use the same framework and module configuration as during an actual game. In summary,

- we use 4 cognition and 2 motion threads,
- we enable real-time scheduling for both threads and prioritize the motion over the cognition cycle,
- we order the update methods of individual modules by their decreasing number of dependents, and
- we do not use CPU core pinning.

	Cognition	
p=0.01	8.918 ms	(−31.05 %)
avg	10.56 ms	(−34.36 %)
p=0.99	13.65 ms	(−30.53 %)
max	14.74 ms	(−33.75 %)
98% range	4.728 ms	(−29.52 %)

Table 6.6: Comparison of cognition metrics to reference measurements.

	motion non-preview		motion preview	
p=0.01	0.8100 ms	(−9.969 %)	1.425 ms	(−5.171 %)
avg	0.8832 ms	(−8.435 %)	1.566 ms	(−3.777 %)
p=0.99	1.001 ms	(−7.268 %)	1.795 ms	(−2.178 %)
max	1.131 ms	(−4.315 %)	1.809 ms	(−4.134 %)
98% range	0.1915 ms	(+6.211 %)	0.3693 ms	(+11.39 %)

Table 6.7: Comparison of motion metrics to reference measurements.

We show the results in tables 6.6 and 6.7. For cognition, the average execution time decreases by 34.36 % to 10.56 ms while the execution trace in figure 6.8 shows that the framework does not even use utilize all four CPU cores simultaneously. Furthermore, the 98 % range of the execution times reduces significantly and results in much more stable

execution times. Additionally, both the 0.99 percentile and the maximum execution time are below the cognition deadline of 33 ms and ensure a stable operation without any deadline misses during our evaluation.

For motion, the average execution times for the non-preview and preview frames decrease by 8.435 % and 3.777 %, respectively. In this case, the improvement is less because many modules execute sequentially and the communication overhead reduces the benefits of two threads even further, which also increases the 98 % range slightly compared to the reference measurements. However, this is negligible because both the 0.99 percentile and the maximum execution time are significantly lower than the deadline of about 12 ms and they are lower than during the reference tests.

Chapter 7

Conclusion and Outlook

The goal of this thesis was to improve the multi-core usage of the current Nao Devils framework in order to exploit the potential of the upgraded CPU in the new NAO version 6. Therefore, we first analyzed the current framework architecture and showed possible starting points to parallelize the existing code. We evaluated the parallel execution of independent modules as the best suitable solution and defined a list of requirements for the new implementation. After that, we examined different approaches for parallel programming and compared their advantages and disadvantages. Based on the results, we decided to use Taskflow, which is a user-space library that allows to execute arbitrary tasks with dependencies using multiple threads and showed to be applicable for our purpose. We integrated the library into the framework and explained the details on that including the generation of the task graph, the changes concerning the inter-thread communication, and the required debug adjustments. From the perspective of other framework developers, we focused on an easy-to-use and dynamic approach that transparently reduces the execution time with minimal configuration effort and minimal changes to the existing code. Furthermore, using the execution trace feature, developers can easily monitor the module scheduling in real-time and investigate the reason of possible performance issues. In the past, the exact execution sequence of modules was unknown and hard to track without further understanding of the framework.

Finally, we compared the performance of the improved framework to the old one. Therefore, we created a realistic evaluation setup that allows to measure the performance as reproducible as possible and tested different configurations in different scenarios. In summary, we measured a reduction of the average execution time by 34 % for the cognition cycle and by 3.8 % to 8.4 % for the motion cycle and confirmed the compliance with the deadlines. Additionally, both cycles currently neither do exploit their available execution time nor do they provide enough parallel workload to utilize all cores simultaneously. That offers great potential for future framework modules that use more complex algorithms, which can improve the overall performance of the robot during the games. Before the

multi-core improvements developed in this thesis, the addition of further modules was only possible because of the increased single-core performance compared to the NAO version 5 but was very limited as well.

Nevertheless, the evaluations showed that the work-stealing based scheduling is not optimal under all circumstances and further improvements may be necessary especially under high-load scenarios or when considering the execution order of update methods of the same module. Furthermore, the NAO's GPU is currently unused and may be considered for execution in the future as well. Independently, the processing of audio for speech recognition and sound localization will also become more important in the future. The SPL rules for the year 2020 state that during a RoboCup competition, the robots must recognize whether the whistle was blown on their own field and at which position the referee was standing. Furthermore, the whole league moves towards more human communication instead of WLAN. Consequently, the speech recognition of commands by the referee is planned for the future as well [24]. Therefore, the Nao Devils team plans to add an additional audio cycle. Since the architecture is designed flexible, the number of cycles can be extended easily. However, then, the number of motion, cognition, and audio threads needs to be reconsidered to ensure optimal performance.

List of Figures

List of Tables

Listings

Bibliography

[1] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. "The design of OpenMP tasks". In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (2008), pp. 404–418.

[2] Robert D Blumofe and Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[3] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[4] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] The Clang Team. *OpenMP support - Clang 9 documentation*. 2019. URL: `https://releases.llvm.org/9.0.0/tools/clang/docs/OpenMPSupport.html` (visited on 05/22/2020).

[6] Brian Gerkey. *Why ROS 2?* URL: `https://design.ros2.org/articles/why_ros2.html` (visited on 05/22/2020).

[7] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. "Cpp-taskflow: Fast task-based parallel programming using modern c++". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 974–983.

[8] Intel Corporation. *Intel Atom Processor E3800 Product Familiy Datasheet*. Oct. 2018. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/atom-e3800-family-datasheet.pdf` (visited on 06/22/2019).

[9] Vasudevan Jagannathan. *Blackboard architectures and applications*. Elsevier, 1989.

[10] Gregor Jochmann, Sören Kerner, Stefan Tasse, and Oliver Urbann. "Efficient multi-hypotheses unscented kalman filtering for robust localization". In: *Robot Soccer World Cup*. Springer. 2011, pp. 222–233.

[11] Ralph E Johnson and Brian Foote. “Designing reusable classes”. In: *Journal of object-oriented programming* 1.2 (1988), pp. 22–35.

[12] Alexey Kukanov and Michael J Voss. “The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks”. In: *Intel Technology Journal* 11.4 (2007).

[13] The Linux Foundation. *Real-Time Linux*. URL: `https://wiki.linuxfoundation.org/realtime/start` (visited on 06/19/2019).

[14] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. “Exploring the performance of ROS2”. In: *Proceedings of the 13th International Conference on Embedded Software*. 2016, pp. 1–10.

[15] Microsoft Corporation. *Enable OpenMP support*. 2019. URL: `https://docs.microsoft.com/en-us/cpp/build/reference/openmp-enable-openmp-2-0-support?view=vs-2019` (visited on 05/22/2020).

[16] Ingo Molnár. *This is the CFS scheduler*. URL: `http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt` (visited on 07/06/2020).

[17] Nao Devils Team. *Code Release 2019*. 2019. URL: `https://github.com/NaoDevils/CodeRelease/tree/CodeRelease2019` (visited on 03/01/2020).

[18] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2013. URL: `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf` (visited on 05/31/2020).

[19] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[20] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[21] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O’Reilly Media, Inc., 2007.

[22] RoboCup Federation. *A Brief History of RoboCup*. URL: `https://www.robocup.org/a_brief_history_of_robocup` (visited on 12/19/2019).

[23] RoboCup Federation. *Objective*. URL: `https://www.robocup.org/objective` (visited on 12/19/2019).

[24] RoboCup Technical Committee. *RoboCup Standard Platform League (NAO) Rule Book*. URL: `https://collaborating.tuhh.de/HULKs/robocup_tc_public/raw/master/SPL-Rules_2020.pdf` (visited on 07/13/2020).

[25] Thomas Röfer, Tim Laue, Andreas Baude, Jan Blumenkamp, Gerrit Felsch, Jan Fiedler, Arne Hasselbring, Tim Haß, Jan Oppermann, Philip Reichenberg, Nicole Schrader, and Dennis Weiß. *B-Human Team Report and Code Release 2019*. URL: `https://github.com/bhuman/BHumanCodeRelease/raw/coderelease2019/CodeRelease2019.pdf` (visited on 04/04/2020).

[26] Thomas Röfer, Tim Laue, Jesse Richter-Klug, Maik Schünemann, Jonas Stiensmeier, Andreas Stolpmann, Alexander Stöwing, and Felix Thielke. *B-Human Team Report and Code Release 2015*. URL: `https://github.com/bhuman/BHumanCodeRelease/raw/coderelease2015/CodeRelease2015.pdf` (visited on 05/26/2020).

[27] Ingmar Schwarz, Oliver Urbann, Aaron Larisch, and Dominik Brämer. *Nao Devils Team Report 2019*. 2019. URL: `https://github.com/NaoDevils/CodeRelease/blob/CodeRelease2019/TeamReport2019.pdf`.

[28] SoftBank Robotics. *Technical overview - Aldebaran 2.8.7.0 documentation*. URL: `http://doc.aldebaran.com/2-8/family/nao_technical/index_dev_naov6.html` (visited on 02/19/2020).

[29] Oliver Urbann, Simon Camphausen, Arne Moos, Ingmar Schwarz, Sören Kerner, and Maximilian Otten. "A C Code Generator for Fast Inference and Simple Deployment of Convolutional Neural Networks on Resource Constrained Systems". In: *arXiv preprint arXiv:2001.05572* (2020).

[30] Oliver Urbann, Ingmar Schwarz, and Matthias Hofmann. "Flexible linear inverted pendulum model for cost-effective biped robots". In: *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*. IEEE. 2015, pp. 128–131.